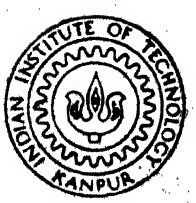
FRONTEND OF A SISAL COMPILER

by

RIYAZ V. P



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MARCH, 1993

THE FRONTEND OF A SISAL COMPILER

A thesis submitted
in partial fulfilment of the requirements
for the degree of
Master of Technology

by
RIYAZ V. P.

to
THE DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
KANPUR - 208 016

MARCH 1993

Certificate

Certified that the work contained in this thesis titled **"The Front-End of a SISAL Compile** has been done by **Riyaz V. P.** (Roll No. 9111123) under our supervision and it has not be submitted elsewhere for a degree.

Dr. Rajat Moona
Asst. Professor
Dept. of Computer Sc. & Engg.
I. I. T. Kanpur

Dr. S. K. Aggarwal
Asst. Professor
Dept. of Computer Sc. & Eng
I. I. T. Kanpur

31 March, 1993

Acknowledgements

I am extremely grateful to Dr. Sanjeev Kumar and Dr. Rajat Moona for their valuable guidance at all stages of this work. I greatly benefited from the frequent meetings with them. They helped me get around all my problems and answered all my questions (even the silly ones!).

Thanks are due to all my friends who made my brief stay at IIT/K a memorable one. It is impossible to list all the names here, but Apu, Sajith and George deserve special thanks for the great "dining table discussions" on everything under (and beyond) the Sun.

Riyaz V. P.

Acknowledgements

I am extremely grateful to Dr. Sanjeev Kumar and Dr. Rajat Moona for their valuable guidance at all stages of this work. I greatly benefited from the frequent meetings with them. They helped me get around all my problems and answered all my questions (even the silly ones!).

Thanks are due to all my friends who made my brief stay at IIT/K a memorable one. It is impossible to list all the names here, but Apu, Sajith and George deserve special thanks for the great "dining table discussions" on everything under (and beyond) the Sun.

Riyaz V. P.

Abstract

The primary goal of this project is to develop a compiler frontend that can serve as a common module in SISAL compilers for a range of target machines. The target architectures might include dataflow, multithreaded architectures, vector processors, systolic arrays and even sequential machines.

SISAL (Streams and Iteration in a Single Assignment Language) is a general purpose, single assignment, applicative language designed for efficient execution on a variety of parallel machines. The single assignment property along with strict functional behaviour makes SISAL highly suitable for parallel programming.

The compiler frontend comprises the lexical analyzer, parser, semantic analysis phase and IF1 graph generator. IF1 (Intermediate Form 1) is a hierarchical graph language based on *acyclic graphs*. IF1 cleanly decouples the frontend of the compiler from the architecture specific backend (code generator). Therefore, the IF1 graph generator can be used in SISAL compilers targetted to different machines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Recent years have seen the advent of many high speed computing systems. These highly parallel machines are a difficult proposition for programmers and compiler writers alike. Programming these systems efficiently is an onerous task. The crux of the problem is in the detection of inherent parallelism in programs.

The job of a programmer is to transform an abstract problem specification into a concrete implementation. Unfortunately, in conventional programming languages, the problem specification and its implementation are inextricably intertwined that it is extremely difficult for the programmer to make a design decision about one without disturbing the other. This problem is even more difficult in a parallel programming environment as the coordination of parallel processes is a very tedious job.

Conventional languages offer little assistance to the programmer in dealing with concurrency. Nor do they help the compiler in the detection of parallelism. In a program, one might encounter many instances where it appears as though two pieces of code are independent; but one cannot be absolutely certain. In all of these cases, the code must be sequenced in order to ensure correctness. In short, the traditional languages are oriented toward sequential processing and are not well-tuned for parallel computing.

An alternative approach for programming parallel machines is to use concurrent versions of conventional languages. These languages provide special constructs by means of which the programmer can explicitly tell the compiler which parts of a program can safely execute in parallel. But the parallelism is possible only among the various independent modules; within

those modules, code has to be sequenced. Smart compilers may be able to search and find some of the unspecified parallelism, but the amount of parallelism they can detect is limited. Moreover, such compilers are not so widely available.

Programming in conventional languages or parallel versions of the same has stifled many opportunities for parallel execution—what we need is a language with no (or perhaps, few) sequential connotations. Ideally, a parallel programming language must have the following properties:

- *It must shield the programmer from such details of the machine as the number and the speed of the processors, the topology of the communication network etc.*
- *Parallelism must be implicit in its semantics.*
- *It must automatically ensure determinancy.*

A single-assignment, applicative language such as SISAL (Streams and Iteration in a Single Assignment Language) [McG85] seems to be a good choice as a parallel programming language. The following points support this view.

Firstly, because of the single-assignment property, the language lends itself naturally to parallel execution, except where sequencing is enforced by data dependencies. The parallelism is implicit in SISAL programs and need not be explicitly specified by the programmer. The single-assignment rule states that once an identifier is assigned a value in an environment (scope of access), it cannot be altered within that environment. This ensures that once an identifier has a value, that value can be sent to all operations requiring the value of that identifier. Identifiers just provide a notational convenience for naming values and therefore every use of a name in an environment refer to the same value.

Secondly, SISAL does not place artificial constraints on the evaluation order. The functional behaviour automatically guarantees determinate results regardless of the execution order (this is known as the Church-Rosser property). There is no need for the programmer to ensure determinancy using special synchronization primitives.

Thirdly, it simplifies a great deal of the work done by the compiler. This does not mean that it immediately solves all the translation issues, in fact, it does not even address all of them.

But, because of the single-assignment rule and the prohibition of aliasing and side effects in functions, it does simplify the global data flow analysis needed for most of the code optimization techniques. Imperative languages have a memory-based model of computation. In other words, the programs are made up of instructions that modify the variables in memory. In general, the variables are widely accessible to many parts of a computation. Therefore, it is difficult for an automatic analysis software to discover which program segments can be safely executed in parallel. The SISAL definition takes care of this problem by insisting on a more disciplined approach to programming.

And fourthly, SISAL is strongly oriented toward scientific computing, and this is the area where parallel machines are employed most. Unlike most functional languages, SISAL uses infix notation for the usual arithmetic operations. It supports iteration and has a rich set of in-built functions for manipulating arrays.

1.2 Goal of this Project

The primary goal of this project is to develop a compiler frontend that can serve as a common module in SISAL compilers for a range of target architectures. The target machines might include data flow architectures, multithreaded architectures, vector processors, systolic arrays and even sequential computers.

Essentially, the frontend consists of a *lexical analyzer* which breaks the source program into a stream of tokens, a *parser* which checks if the source program is syntactically correct, a *semantic analyzer* which checks if the program is semantically valid, and an IF1 *graph generator* which generates an IF1 graph representation [Ske85a] of the program. The actual implementation has all of these phases working together—side by side.

IF1 (Intermediate Form 1) is a hierarchical graph language that decouples the frontend of the compiler from the architecture specific backend (code generator). Both SISAL and IF1 were developed by researchers at Lawrence Livermore National Laboratory (LLNL), Digital Equipment Corporation (DEC), Colorado State University (CSU) and the University of Manchester (UM).

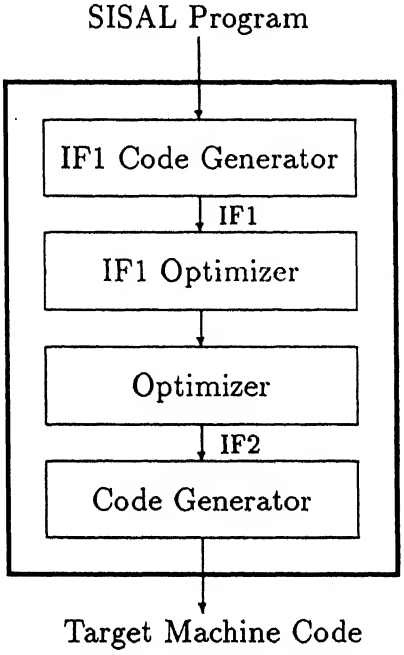


Figure 1.1: *Structure of a SISAL compiler*

1.3 Basic Structure of a SISAL Compiler

Figure 1.1 shows the basic structure of a SISAL compiler. The lexical analysis and parsing phases are not shown in the figure.

Essentially, the compiler consists of a language specific frontend and a machine specific backend or code generator. In this project, the first module i.e., the translator from SISAL to IF1 has been implemented. The IF1 generator takes a SISAL compilation unit (see next chapter) and translates it into a set of IF1 graphs, one for each SISAL function in the unit. The output of our translator is a text file composed of a sequence of lines, each line representing an "IF1 entity" (see Chapter 3).

The second phase is an optimizer for IF1 graphs; most of the classical code optimization techniques can be easily applied to IF1. This part of the compiler could be implemented as an immediate continuation of this project or it could wait till the code generator is also written. Since IF1 cleanly decouples the frontend from the backend, the optimizer module can be easily plugged into the compiler at a later time. In fact, this module may viewed as a part of the frontend and can be used as a common block in SISAL compilers for different machines.

The IF1 optimizer module is followed by a machine specific optimizer. The output of this optimizer—shown as IF2 in the figure—may be machine dependent. The last phase viz. the code generator is the real heart of the compiler.

1.4 Organization of this thesis

The remainder of this thesis is organized into four chapters. The next chapter gives an overview of the SISAL language. It touches upon only the most relevant features of the language and is therefore not a substitute for the language manual [McG85].

In the third chapter, we present an overview of the IF1 language. More details about IF1 may be found in [Ske85a].

We describe the main content of this project in the fourth chapter. We take each SISAL programming construct and set about describing how it is translated into its equivalent IF1 representation. The chapter winds up by addressing the issue of testing the translator.

In the concluding chapter, we look at the various possible continuations of this project. First we consider some simple tools that could be developed to aid the IF1 user. These include an IF1 interpreter and a program for generating the pictorial form of IF1 graphs. We then assess the quality of the code generated by our translator and see how it can be improved. Finally, we conclude this thesis with a summary of what we have done so far and what could be done to improve it.

There are four appendices. Appendix A and B give the syntax rules for SISAL and IF1 respectively, in an extended BNF notation. Appendix C contains a sample SISAL program and its corresponding IF1 representation. Appendix D is the User's Manual for our IF1 generator.

Chapter 2

An Overview of SISAL

SISAL is a general purpose, single assignment, applicative language designed for the efficient execution of programs on a range of parallel computers. Essentially, SISAL is an offspring of Ackermann and Dennis' dataflow language VAL [McG82]. In this chapter, we survey the most germane features of SISAL. It is not our intention to give a complete description of the language here, rather we will concentrate on the interesting and noteworthy aspects of SISAL. In particular, we will lay emphasis on the unusual aspects about data types, concepts like value orientation and single assignment rule, prohibition of side effects and aliasing in functions, iterative and parallel versions of loops, handling of I/O without side effects and error handling. The reader will find a detailed description of the language elsewhere [McG85]. The syntax of SISAL is given in Appendix A. The description below is fairly informal with most of the ideas expressed through examples.

2.1 Some Basic Features

Before we set about describing the more interesting features, let us look at a few aspects which though not unique to SISAL are noteworthy.

- SISAL *supports separate compilation.* A SISAL program is made up of modules that can be compiled separately. Each such module is called a *compilation unit.* Each compilation unit is a collection of SISAL functions. A compilation unit may contain invocations of functions defined in other units. However, since SISAL advocates strong type checking, the type of the function should be known to the compiler when its invocation is encountered. This is

```
type int_vector = array[integer];
type real_matrix = array[array[real]];
type stud_rec = record[
                    name : array[character];
                    roll_no, age : integer
                ];
type bool_stream = stream[boolean];
type binary_tree = union[
                        not_empty : tree_node;
                        empty : null
                       ];
type tree_node = record[
                    leftchild, rightchild : binary_tree;
                    data : character
                   ];
```

Figure 2.1: *Examples of Structured Types in* SISAL

done by declaring the prototypes of all external functions using the **global** clauses. Not all functions defined in a compilation unit are available to other units. An "exportable" function is identified using the **define** clause.

- SISAL *programs are block-structured.* Function definitions may be nested within other functions. The scope rules for nested functions are similar to PASCAL. Some expressions like **let** and **for** permit block structuring within the function body.

- SISAL *allows recursion.* In SISAL, apart from **for** expressions, recursion allows repetitive execution of the same code. It may be mentioned here that VAL, the ancestor of SISAL does not support recursion. SISAL also supports mutually recursive functions with the help of forward declaration of function headers. A forward declared function must be defined before the end of its scope.

2.2 Data Types

SISAL is a strongly typed language. The elementary (scalar) data types in SISAL are similar to those found in most conventional languages. The scalar types are **boolean**, **character**, **integer**, **real**, **double_real** and **null**. Of these only **null** seems to be unfamiliar. It is used in conjuntion with the discriminated union data structure. The constructed (structured) data types in SISAL include **array**, **stream**, **record** and the discriminated **union**. See Figure 2.1 for examples of structured data types. There is a rich set of operations predefined on each of these types.

The principal difference between SISAL types and types in other languages are in the definition and use of constructed types, in type-checking rules and in the existence of error values within every type.

2.2.1 Arrays

SISAL arrays are unusual because the array bounds (or sizes) are *not* part of the type definition. Only the component type of the array is specified in the definition. Most of the array operations in SISAL are defined in such a way that the possibilities for concurrency are enhanced. For example, in an array constructor (also called array generator), all elements of an array can be specified simultaneously, thus allowing all of the evaluations of array elements to proceed in parallel. Some example array operations are shown in Figure 2.2.

2.2.2 Streams and their use in I/O

A stream is an ordered sequence of values of the same type. Streams are like arrays in that they can grow arbitrarily large at run time. That is, the size of a stream is not a part of its definition. Unlike arrays, a stream does not permit random selection of its components; only the first element can be selected. A stream is more like a queue allowing addition of elements to take place at one end and their removal from the other end. We digress here for a moment to point out a restriction that the SISAL definition imposes on all operations: No operation is allowed to modify its inputs. When we talk of an operation that removes an element from a stream, what we mean is that the operation creates a new stream with its front element removed, the input stream remaining intact. This is in fact a consequence of the functional behaviour exhibited by

In the examples, *vector* is a type-name denoting array[real]; A, B and C are arrays of type *vector*; I and J are values of type integer; P and Q are values of type real.

- array *vector*[ ] creates an empty array of type *vector*.
- array *vector*[$I : P, Q$] creates an array of type vector with two elements P and Q. The index of P is I. Q immediately follows at index $I + 1$. Let us call this array A for reference in the examples below.
- array_size(A) is 2, since A contains two elements.
- array_liml(A) returns I, the lower limit of A.
- array_limh(A) returns $I + 1$, the higher limit of A.
- array_addl($A, 3.14$) results in a new array with three elements, $3.14, P, Q$. A new element is added at the lower index. The lower limit is still I. Call this array B, for future reference.
- The selection operation $B[I]$ returns 3.14 and the operation $A[I]$ returns P.
- $B \,||\, A$ combines B and A producing an array with the five elements $3.14, P, Q, P$ and Q. Let this array be called by the name C.
- $C[I + 3 : 5.22, 6.78]$ produces a new array with the two elements of C at indexes $I + 3$ and $I + 4$ replaced with 5.22 and 6.78 respectively. This array will contain $3.14, P, Q, 5.22$ and 6.78

See the manual for a precise semantics of these operations.

Figure 2.2: *Examples of Array Operations*

In the examples below, *list* is a type-name denoting stream[integer]. G and H are streams of type *list*. I and J are integer values.

- stream *list* [ ] creates an empty stream.
- stream *list* [$22, 32, 14, 65$] creates a stream with four elements: $22, 32, 14$ and 65. Call this stream G.
- stream_first(G) returns 22.
- stream_rest(G) returns a stream with three elements: $32, 14$ and 65. That is, a stream containing all the elements of G except the first. Call this stream H.
- stream_size(H) returns the size of H, 3.
- stream_empty(H) returns false because H is not empty.
- $G \,||\, H$ combines G and H to yield a new stream containing the seven elements, $22, 32, 14, 65, 32, 14$ and 65.

Figure 2.3: *Examples of Stream Operations*

SISAL operations. This property of SISAL operations will be elaborated in a later section on *value orientation.*

An important use of streams is in the handling of I/O. Input and output pose a serious problem in functional languages because read and write operations normally violate the functional behaviour. A read operation not only returns the required values, it also produces the side effect of updating a global state so that later reads will pick up the next set of data in the input. Writes have the same problem. If two SISAL functions need to write, they must have some form of synchronization to ensure deterministic results. SISAL adopts the concept of streams for handling I/O without side effects.

The primary operations defined on a stream are adding and removing elements from an end. In certain ways, the stream is similar to the Unix pipe. The key thing about streams is in the way they are built and used. In all the other SISAL types, an object must be completely built before any function can access any portion of the object. For example, all elements of an array must have values before any function can begin to access elements. Streams are more flexible; if a function F produces a stream to be used by function G, SISAL allows F to start giving the

```
type my_rec = record [
                        name : array [ character ] ;
                        roll_no : integer ;
                        ] ;
```

- **record** *my_rec* [*name* : "Riyaz"; *roll_no* : 9111123] creates a record of type *my_rec*. Call this record *R*.
- *R.roll_no* yields 9111123.
- *R* **replace** [*roll_no* : 3211119] creates a new record with the name field same as that of *R*, but *roll_no* changed to 3211119.

Figure 2.4: *Examples of Record Operations*

front end of the stream to G while F continues to build the remainder. In effect, a stream defines a natural form of pipelined communication from the producing function to the consuming one.

This notion of streams allows programs to represent basic forms of input and output in a safe manner without requiring explicit synchronization. Input and output can be achieved by passing streams into and out of the main function. Input streams correspond to values received by the program in response to read requests; output streams correspond to write actions.

Figure 2.3 gives some examples of stream operations.

2.2.3 Records

Record construction is patterned after array construction to improve concurrency. All the field values of a record can be computed concurrently. The operations on records include selecting a component by field-name and altering the value of a field[1]. Examples of operations on records are shown in Figure 2.4.

2.2.4 Unions

The last type constructor is the union. This permits the construction of discriminated union types—values of this type appear to be of different types at different points of execution. A union consists of a number of tag-names just as a record contains field-names. Each tag-name

[1] Once again, the reader is reminded that the input record remains intact; a new record with the specified field value is constructed.

```
type aunion = union [
                    int : integer ;
                    bool : boolean ;
                    nothing : null ;
                    ] ;
```

- **union** *aunion* [*int* : 7641] creates a union with tag *int* and element value 7641. Call this union U.
- is *bool* U yields **false** because the tag of U is *int*, and not *bool*.
- **union** *aunion* [*nothing*] creates a union with tag nothing. Since the tag is of type null, no component value is specified—it is **nil** by default.

See Figure 2.8 for an example of **tagcase** expression.

Figure 2.5: *Examples of Union Operations*

has a type associated with it. To construct a union type, we select one tag-name and tie it to an expression which has the same type as that of the tag-name. The union data object so constructed has the the value of the specified expression. The **tagcase** expression which allows conditional execution depending on the tag of a union object will be discussed later. Another operation defined on union is the testing of its tag. Examples of some union operations are given in Figure 2.5.

2.2.5 Type Checking in SISAL

SISAL imposes strong structural type checking. In SISAL two types are equivalent if they represent identical structures; the names bound to the types have no influence on the type analysis. This method of type checking has an advantage. It eliminates the need for exporting and importing type definitions from one compilation unit to others. Automatic type conversions are *never* made by the compiler. The following rules govern the type conformance checking algorithm.

1. *Two basic types conform if they are the same.*
2. *Two array (stream) types conform if their base types conform.*

3. *Two record (union) types conform if the number of components is the same and, the names of corresponding components are the same and their types conform. The order in which the components appear must also be the same.*

4. *A user type conforms with the right hand side of its definition.*

As seen in the example of Figure 2.1, SISAL allows mutually recursive types (*binary_tree* is defined in terms of *tree_node* and vice-versa). The type checking algorithm based on the above rules is described in Chapter 4. We shall see that this algorithm is capable of handling cyclic types like the *binary_tree* of our example.

As a note, it is not mandatory to declare the types of value-names (value-names are SISAL's answer to variables in traditional languages, see next section). The compiler can deduce the types of value-names from the types of expressions assigned to them. As another implementation note, all type checking in SISAL can be done at compile time.

2.3 The Value Oriented Philosophy

SISAL follows a value oriented style as opposed to the variable orientation of traditional languages. Conventional languages have a memory based model of computation. These languages rely on concepts like variables and memory updating. A variable denotes a memory location rather than the value it contains. Data objects are modifiable or mutable by program statements as the computation proceeds. This highly unrestricted style of programming encouraged by traditional von Neumann type languages leads to the automatic detection of parallelism extremely difficult.

In SISAL the above concepts are replaced by a value system in which every object is immutable. Data objects in SISAL are therefore mathematical values in the strictest sense. At first glance this restriction may seem too severe to programmers. Fortunately, it is not so. The idea of binding values to identifiers is still available in SISAL. But it is important to note that identifiers are value-names rather than variables. We thus come to the most important characteristic of SISAL: *single assignment.* The single-assignment rule states that

> *Once an identifier is bound to a value, the binding remains in force for the entire scope of access to that identifier*

Operations on structured objects are most effected by the value orientation. Just like any other data object, arrays and records are also treated as mathematical values. Hence they can never be modified. The only option is to build a new array or record that has the same values in all of the old positions except for the particular element that is to be changed. This rule is enforced by requiring that every identifier-value binding be made to a full identifier and not to a field or subscript position. For example, a binding beginning with '$A[I] := \ldots$' is illegal.

If the above concepts are implemented naively, it will lead to a lot of unnecessary copying of large data structures like arrays. Though the single assignment rule must be strictly adhered to at the programmer's level, a good implementation should avoid copying arrays as far as possible. Avoiding copies is not only conservative of space but also saves time. The reader might be confused with the above seemingly contradictory statements. What we are trying to say is simply this: from a programmer's point of view, an operation behaves as if it creates a new array whereas in the actual implementation it need not always be the case. The compiler should copy arrays only when it is absolutely necessary to do so. This problem of *copy avoidance* or *copy optimization*, as it is usually known, is an important issue in single-assignment language implementation.

2.4 Expressions and Functions

In SISAL expressions and functions take the place of statements and procedures just as value-names substitute for variables. There are no statements in the conventional sense. All active (nondeclarative) language features are functions; they use values provided by the current execution environment and have the sole effect of producing a set of result values[2].

SISAL functions enjoy freedom from side effects. It is important to note this because the prohibition of side effects and aliasing plays a major role in simplifying the automatic program analysis. All parameters are passed by value and their evaluations can execute concurrently. Parameters are bound to values at the call site and the function cannot rebind those values. A function has no access to global identifiers (except type-names and other function-names).

SISAL has four expressions that reflect the notion of control flow in conventional languages: **let**, **if**, **tagcase** and **for**. Unlike their counterparts in most languages, these are expressions

[2]SISAL functions may return more than one value.

```
let
   radius : real;                    % declare radius
   pi, pi2 : real := 3.1416, 6.2832; % declare and define pi and pi2
   radius         := 100.0;          % define radius
   sq_rad         := radius * radius % define sq_rad
in
   pi * sq_rad, pi2 * radius % the result values of let expression
end let
```

Figure 2.6: *Example of LET expression*

```
if a > b then a - b
elseif a < b then b - a
else a + b
end if
```

Figure 2.7: *Example of IF expression*

instead of statements.

2.4.1 Let Expression

The **let** expression allows the programmer to name intermediate results of a computation prior to computing the final result value(s). The **let** consists of two parts: the name-value binding part and an expression part separated by the keyword **in** (see Figure 2.6). Each name introduced in the binding part must be defined exactly once. It is optional to declare the type of a name. The right hand side of a binding may refer to a previous binding. The expression part consists of an expression of arbitrary arity which is evaluated and returned as the result of the **let** expression. The scope of the bindings is the entire **let** expression.

2.4.2 If Expression

The **if** expression (see Figure 2.7) in SISAL is analogous to the *if statements* in other languages. Like its counterpart, the **if** expression uses a Boolean value to determine which of several possible result values to produce. Unlike its counterpart, the result clauses must be expressions (of the

Let U be an expression of the following **union** type:

```
type a_union = union [
                    a, b, c : integer ;
                    d, e, f : real ; g : character;
                    ] ;
```

In the following tagcase expression P, Q and R are value-names of type integer. A is a value-name of an array type. C is a value-name of type character. The tagcase expression returns two values, one an integer and the other a boolean.

```
tagcase x := U
tag a,c : x, A[x] > x                  % x is an integer in this arm
tag d : integer(P + x), x = 3.14       % x is a real in this arm
tag g : 26, (x > 'a') & (x < 'z')      % x is a character in this arm
otherwise : trunc(P + Q), Q > R % x is not defined in this arm
end tagcase
```

Note that the otherwise-clause is required because the tags b, e and f are absent in the tag lists. The reason for not allowing x to be visible in the otherwise-clause should also be clear from the above example—the left out tags have different types.

Figure 2.8: *Example of TAGCASE expression*

same type) instead of statements. An arbitrary number of **elseif** clauses are permitted making the **if** expression akin to a *case statement*. To ensure that the expression will always return some value, the **else** clause must always be present. The **if** expression introduces no new name-value binding. All outer value-name scopes pass into an **if** construct.

2.4.3 Tagcase Expression

The **tagcase** expression (Figure 2.8) permits access to values of **union** type. The tag field is interrogated to discover the union-value's true type and a multiway branch is used to select the appropriate result expression. The value associated with the union can be assigned a name. The type of this name is different in different arms of the expression. Like the **if** expression, all arms of a **tagcase** must generate the same type of expression.

```
for initial
    i := 1;
    s := 0.0;
while i <= n repeat
    i := old i + 1;
    t := old s * a[i] + 1;
    s := i + t
returns
    value of tree sum s
    value of catenate array [1 : i]
    array of i
end for
```

Figure 2.9: *Example of FOR expression*

2.4.4 For Expression

The **For** expressions correspond to the *loops* found in other languages. There are two versions of the **For** expression: the iterative or sequential version also called the non-product form and the parallel version also called the product form.

Iterative Version

The iterative version of **for** expression has four basic parts, each identified by a special keyword: *initialization section* (prefixed by the keyword **initial**), *repetitive section* (prefixed by **repeat**), *termination test* (prefixed by **while** or **until**), and a list of *result clauses* (prefixed by **returns**). See Figure 2.9.

The initialization section is used to create local identifiers (loop-names) that will be used throughout the **for** expression and to give them initial values. The **initial** section is treated as the first pass of the loop. The repetitive section is used to specify repeated bindings to loop-names. The **repeat** section may be viewed as a function that returns the new value binding for each loop-name. The value bound to a loop-name in the previous pass is obtained by prefixing the loop-name by the keyword **old**. The termination test may appear either before or after the **repeat** body, indicating when the test is to be performed. Loop-names with **old** modifier must

not appear in termination tests occurring before the **repeat** body.

The result clause has many unusual features. When a loop terminates, the simplest way to return a result would be to say "**value of** ⟨*expr*⟩", which is the value of ⟨*expr*⟩ with the last binding of loop-names[3]. Using "**array of**" or "**stream of**" instead of "**value of**" would result in values of ⟨*expr*⟩ for each binding of loop-names to be collected and returned as an array or stream. Another interesting option is "**value of sum** ⟨*expr*⟩". This returns the sum of the values of ⟨*expr*⟩. In addition to **sum**, permitted reduction operators include **product**, **least**, **greatest** and **catenate**. SISAL also allows the associativity of these operators to be specified. The associativity of an operator may be **left**, **right** or **tree**. Certain values from the sequence of values of ⟨*expr*⟩ can be excluded from the result calculation. This is done by specifying a Boolean condition (masking expression) in the result clause which has to be satisfied (or unsatisfied) in a loop pass if the value of ⟨*expr*⟩ in that pass is to be used in the result computation. No loop-name with **old** prefixe is allowed in a result clause. However an **old** prefix may be used for the result clause as a whole.

Parallel Version

SISAL includes a parallel form of **for** expression. This expression is useful when the same set of actions need to be performed over a specific range of values.

The parallel **for** expression has three sections: the *range specification section*, the *body* and the *return clauses*. The range specification section may specify a range of integers between two bounds, or a range that runs through the elements of an array or stream. More complex ranges may be specified by using multiple ranges to be **crossed** (outer product) or **dotted** (inner product). See the manual for more details.

The body and result clauses are quite similar to the sequential version. See Appendix C for an example of the parallel **for** loop.

The parallel version of **for** expression highlights the SISAL approach to concurrency. Although the loop bodies can be executed in parallel, the compiler can choose to implement it as a fully parallel operation, a pipelined operation, or even as a sequential loop. This choice depends on the availability of resources like memory and processors.

[3] ⟨*expr*⟩ has a different value in each pass of the loop. The last value in this sequence is used.

2.5 Error Handling in SISAL

In the area of error handling, SISAL takes an approach which is unusual as far as conventional languages are concerned. The approach adopted by SISAL is quite similar to the one employed by VAL. In sequential languages, it is a relatively straightforward operation to bring a computation to a halt after an error occurs. But in a more complex environment like the dataflow, such a task is more difficult to do in a way that will permit the programmer to determine what went wrong. Because of this difficulty, SISAL includes a somewhat unusual semantics for allowing programs to continue safely in the event of an error.

When an operation results in an exception, it produces a special error value of the appropriate type. The value **error** is a proper element of every SISAL type. This value could be produced as a result of an arithmetic error or a control flow error. For example, addition of two integers might cause an overflow. In SISAL, the value of such a computation is **error[integer]**. Similarly, a conditional test in an **if** expression or a **for** expression might produce an error value—**error[boolean]**. If it does, the **for** or **if** expression produces error values of the appropriate type as results. SISAL also provides a special function for testing for the presence of these error values, since normal tests would not give the correct answer (**error[boolean]** = **error[boolean]** yields **error[boolean]** and not **true** as one would expect). The **is error** function indicates whether or not the input parameter is an error value. The output of **is error** will *never* be an error value.

Normally, an operation receiving an erroneous input simply produces appropriate error values as output. However, operations on arrays have a more complex semantics. The semantics of these operations allows one to access as many elements as possible in erroneous arrays. We will not discuss this aspect in detail here. Suffice is to say that the main objective of this error handling aspect of SISAL is to retain as much information as possible after an error has occurred. The details can be found in the language manual.

2.6 Summary

From the user's view, the functional style has its good and bad sides. The bad side is that programmers have to adopt a more restrictive discipline of programming. The good side is that

they have a clean and simple environment for expressing parallelism safely, without concern for making synchronization mistakes. Parallelism in SISAL is found not just in the parallel version of **for** loop. If two expressions do not need each other's answers (either directly or indirectly), they can execute simultaneously without the possibility of any conflict or error. This condition is the only rule for identifying concurrency in SISAL. The restrictions imposed to the programming style by SISAL reduces the compiler effort required for program analysis because all dependencies are clearly stated.

Chapter 3

An Overview of IF1

IF1 is a hierarchical graph language developed for use as an intermediate language in compilers for high level data flow languages. IF1 provides a common interface between the frontend and the backend of SISAL compilers for different machines. In this chapter, we present a brief overview of the IF1 language. For the complete syntax of IF1, see Appendix B. The reader may refer to the IF1 language reference manual [Ske85a] for more details.

3.1 The IF1 language

IF1 is a hierarchical graph language based on *acyclic graphs*. A SISAL compilation unit is translated to an IF1 file which is composed of four types of components: *nodes, edges, types and graph boundaries*. Nodes represent operations, edges denote data dependencies, and the type attached to an edge denote the type of data carried by it. A graph boundary encloses a group of nodes and edges. Figure 3.1 illustrates the various components of an IF1 graph. The graph represents the expression $-X + 2 * A[I]$.

An IF1 graph contains a set of nodes interconnected by edges. The boundary of a graph is clearly marked out by what is called a *graph boundary node*. The graph boundary serves as the source and sink of values entering and leaving the graph. The graph boundary contains a set of input and output ports that serve as the interface between the nodes of the graph and the external environment. The graph receives its input values from the outside environment through the input ports. The values computed inside the graph are transported to the outside environment via the output ports. More than one edge may use an input port on the boundary as the source. But two edges are not allowed to use the same output port as the destination.

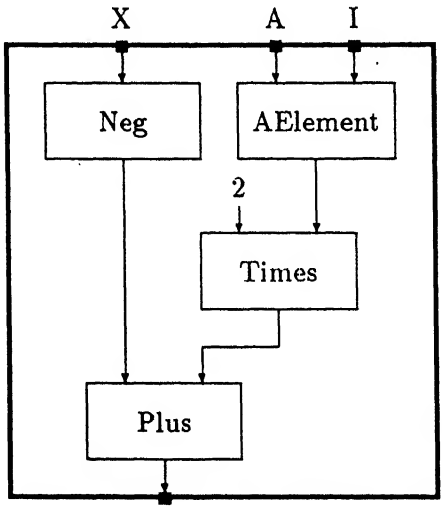


Figure 3.1: *An Example* IF1 *Graph*

In the figure, the large boldface box represents a graph boundary. This graph has three input ports, one for A, another for I and a third for X (the ports are shown as small dark pegs on the graph boundary). The graph has one output port for its result.

The smaller boxes in the figure represent nodes. There are four nodes in the example graph: *Neg*, *AElement*, *Times* and *Plus*. Of these, *Neg* has only one input port; others have two each. All the nodes have one output port each. In general, a node may have multiple input ports and multiple output ports. Values enter a node through its input ports, the node performs its operation, and the results are passed out through its output ports. There can be at most one edge into a node's input port; but more than one edge can emanate from an output port. The nodes shown in the figure are *simple* in the sense that they represent simple elements of computation, such as an arithmetic or Boolean operator, or a simple operation like array selection. IF1 nodes can also be *compound*. Compound nodes represent the structured expressions of SISAL namely, the **if/then/else** expression, the **tagcase** expression, and the different **for** expressions.

A compound node comprises a number of subgraphs with implicit connections between ports on the subgraph boundaries and ports on the compound node. Each subgraph of a compound node defines a part of a structured expression. There are five compound nodes defined in IF1 as shown in Table 3.1.

The directed arcs in Figure 3.1 are edges. Usually, edges have a source node and a destination

Name	SISAL expression
Select	if/then/else
TagCase	tagcase
LoopA	iteration with termination test after one pass
LoopB	iteration with termination test before one pass
Forall	product form of for-expression

Table 3.1: *Compound Nodes*

Character	Meaning
C	comment
G	graph
X	exported function
I	imported function
E	edge
L	literal edge
T	type
N	node
{	start of a compound node
}	end of a compound node

Table 3.2: IF1 *components*

node. Edges connect an output port of the source node to an input port of the destination node. However, in the figure, the edge incident on the first port of the node labelled *Times* does not have a source node. This is because the value carried by this edge is the literal constant "2". Such edges are called *literal edges*. Edges also have types associated with them; this is not shown in the figure.

3.2 IF1 File

An IF1 file consists of a set of *lines* delimited by newline characters. The first non-blank character on each line gives the meaning of the line. Given in Table 3.2 are the permitted first non-blank characters. They have the indicated meanings.

3.3 Nodes

As we have already seen, nodes can be either *simple* or *compound*. Simple nodes have the form:

N ⟨*label*⟩⟨*op*⟩

where the ⟨*op*⟩ is an ASCII string that represents the operation performed by the node. The various operations in IF1 are listed in Appendix B. ⟨*label*⟩ is an integer that numbers the node. This number is used to denote the source and destination nodes of edges.

Usually, simple nodes have a fixed number of inputs. However, there are a few simple nodes that allow variable number of input ports. *ABuild* used for building arrays is an example of a node with variable inputs. It takes an integer for the lower index of the array on the first input port, and component values on ports 2, 3, ... etc. We shall come across examples of many simple nodes when we discuss the translation of simple SISAL operations to the corresponding IF1 nodes.

3.4 Graph Boundaries

An IF1 graph encapsulates a possibly large body of computation. A SISAL function is represented in IF1 by a graph. The input and output parameter lists in the function header determine the input and output ports of its graph boundary node. Graphs in IF1 appear as either function graphs or as subgraphs of compound nodes. Function graphs are at the highest level of the hierarchy; they may contain compound nodes which in turn may contain other compound nodes etc. In an IF1 file, graph boundaries begin with a line containing G (or X, or I) and a type reference. A graph boundary may either represent a function or a subgraph of a compound node. Graph boundaries that represent functions are denoted:

G ⟨*type-reference*⟩ "⟨*name*⟩" % for a local function definition
X ⟨*type-reference*⟩ "⟨*name*⟩" % for a global function definition
I ⟨*type-reference*⟩ "⟨*name*⟩" % for an imported function

The field ⟨*name*⟩ above is the name of the function. The ⟨*type-reference*⟩ field associates a type to a graph denoting a function. Graphs that are subgraphs of compound nodes are not typed (this is denoted by a zero in the type reference field). We shall see more about type descriptors in a later section.

Entry	Code	Param1	Param2
Array	0	base type	not used
Basic	1	basic code (see Table 3.4)	not used
Field	2	field type	next field
Function	3	argument type	result type
Multiple	4	base type	not used
Record	5	first field	not used
Tag	6	tag type	next tag
Tuple	7	type	next in tuple
Union	8	first tag	not used

Table 3.3: *Type Entries*

The graph boundary descriptor for a local or global function is immediately followed by nodes and edges that constitute the body of the function. However, the graph boundary for an imported function merely associates a type descriptor with a function name—no nodes or edges can follow it.

Graph boundary nodes are always labelled zero. All other nodes have labels greater than zero. Each graph begins numbering its nodes from one onward.

3.5 Edges

Explicit data dependencies within a graph are denoted by edges. An edge may be either an ordinary edge with source and destination nodes, or a literal edge which has no source node. Their respective forms are:

E ⟨*src-node*⟩⟨*src-port*⟩⟨*dst-node*⟩⟨*dst-port*⟩⟨*type-reference*⟩
L ⟨*dst-node*⟩⟨*dst-port*⟩ "⟨*value*⟩" ⟨*type-reference*⟩

⟨*src-node*⟩ and ⟨*dst-node*⟩ are the labels of source and destination nodes respectively. ⟨*src-port*⟩ and ⟨*dst-port*⟩ denote the port numbers of the source and destination nodes. The field ⟨*type-reference*⟩ denotes the type of the value carried by the edge. In the case of literal edges, the field ⟨*value*⟩ denotes the value of the literal.

Type	Code
Boolean	0
character	1
double	2
integer	3
null	4
real	5

Table 3.4: *Basic Type Codes*

3.6 Type Descriptors

A type may be viewed as a linked list that associates type constructors with component types and functions to arguments and results. A type descriptor is referenced in edges and graph boundaries for functions. A type descriptor has the following form:

T ⟨*label*⟩⟨*code*⟩⟨*param*1⟩⟨*param*2⟩

⟨*label*⟩ is a positive integer that numbers a type descriptor. It is this ⟨*label*⟩ that is used in the type-reference field of edges and graph boundaries. The other fields (⟨*code*⟩, ⟨*param*1⟩ and ⟨*param*2⟩) take the values indicated in Table 3.3.

3.7 Compound Nodes

Compound nodes contain subgraphs and span many lines. The number of subgraphs in a compound node may be fixed (in LoopA, LoopB and Forall nodes), or may vary (in Select and TagCase node). Each compound node has its own semantics that relate its inputs and outputs to the inputs and outputs of its subgraphs.

The form of a general compound node is:

{
G 0
... $subgraph_0$
G 0
... $subgraph_1$
⋮
G 0

... $subgraph_n$
} $\langle label \rangle \langle op \rangle$ k a_1 a_2 ... a_k

The { marks the beginning of a compound node. Each subgraph begins with a line containing G 0. The } closes the compound node. The $\langle label \rangle$ and $\langle op \rangle$ fields have the same meaning as in simple nodes. k is the length of the association list a_1 a_2 ... a_k. The association list maps the subgraphs that are described in the semantics of each compound node to the subgraphs in the IF1 file.

3.7.1 Implicit Dependence in Compound Nodes

There are no explicit edges to specify how the subgraphs of a compound node are dependent on one another. All such information is implicit in the compound node. Three kinds of implicit dependence can exist within a compound node:

1. *Data dependence between compound node and its subgraphs.* The input values of a compound node are passed to each of its subgraphs. No explicit edges are however used to express this dependence.

2. *Data dependence between subgraphs.* This type of dependence occurs in loops. The values computed by the Init subgraph may be used by the Body subgraph, Test subgraph and Returns subgraph. Similarly the values computed by the Body subgraph may be passed to the Returns subgraph, Test subgraph or back to the Body subgraph.

3. *Control dependence.* This type of dependence occurs in selection and iterative loops. In selection, one of the alternative subgraphs have to be selected, depending on some Boolean condition. In a loop, iteration has to be terminated depending on some Boolean condition.

3.7.2 Classes of Values and Ports

Every value that is passed into or passed out of a compound node subgraph has a *class*. The class designates the value as *imported* value, *loop* value etc. Just as we talk of classes of values, we can also talk of classes of ports on the subgraph boundary. All ports of the same class are numbered contiguously. The semantics of the compound node lays down that ports of a particular class should come before or after ports of some other class. For example, in a LoopA node, ports of

imported values (class K) must precede the ports of loop values (class L). This division of values into different classes is useful in loop analysis, because loop values can be quickly separated out from loop invariants.

In our following discussion of compound nodes in IF1, we shall make use of two tables for each compound node. One table gives the various port classes and their relative ordering. The second table tells us which port classes are allowed for the input and output ports of the different subgraphs.

Now we briefly describe the various compound nodes.

3.7.3 LoopA Node

The LoopA node is used to represent an iterative loop with its termination test after the loop body. A LoopA node has four subgraphs: Init, Body, Test and Returns. The Init subgraph initializes the loop values. The Body subgraph produces new loop values in every pass of the iteration. The Test subgraph determines when the loop is to be terminated. A false value generated by the Test subgraph signals the end of iteration. The Returns subgraph collects the loop values and produces the results of the LoopA node from them. The association list for a LoopA node is:

4 ⟨*init*⟩⟨*test*⟩⟨*body*⟩⟨*returns*⟩
where ⟨*init* is the number of the subgraph corresponding to Init
⟨*test*⟩ is the number of the subgraph corresponding to Test
⟨*body*⟩ is the number of the subgraph corresponding to Body
⟨*returns*⟩ is the number of the subgraph corresponding to Returns

The implicit dependencies in a LoopA node are given below:

1. All inputs to the LoopA node are available to each subgraph on the class K ports.
2. The loop values from the output of the Init subgraph are connected to the inputs of the Body and Returns subgraphs on the class L ports.
3. The loop values from the output of the Body subgraph are connected to the inputs of the Test, Body and Returns subgraphs on the class L ports.

Class	Usage	Port Range
K	LoopA input values	$1 \ldots n_K$
L	loop values	$n_K + 1 \ldots n_K + n_L$
T	derived loop values	$n_K + n_L + 1 \ldots n_K + n_L + n_T$
B	Boolean result of Test	$1 \ldots 1$
R	LoopA results	$1 \ldots n_R$

Table 3.5: *Port assignments for each class in LoopA*

Node/ Subgraph	Input Ports	Output Ports
LoopA	K	R
Init	K	L
Test	K, L, T	B
Body	K, L	K, L, T
Returns	K, L	R

Table 3.6: *Port usage for each subgraph in LoopA*

4. The derived loop values from the output of the Body subgraph are connected to the inputs of the Test subgraph on the class T ports.

5. The results of the Returns subgraphs are connected to the output ports of the LoopA node on the class R ports.

6. When the result of the Test subgraph is false, the LoopA node makes its results available at its output ports.

Tables 3.5 and 3.6 give the port assignments and port usage in a LoopA node.

3.7.4 LoopB Node

The LoopB node is used to represent an iterative loop with its termination test before the loop body. A LoopB node also has four subgraphs: Init, Body, Test and Returns. The Init subgraph initializes the loop values. The Body subgraph produces new loop values in every pass of the iteration. The Test subgraph determines when the loop is to be terminated. A false value generated by the Test subgraph signals the end of iteration. The Returns subgraph collects the

Class	Usage	Port Range
K	LoopB input values	$1 \ldots n_K$
L	loop values	$n_K + 1 \ldots n_K + n_L$
B	Boolean result of Test	$1 \ldots 1$
R	LoopB results	$1 \ldots n_R$

Table 3.7: *Port assignments for each class in LoopB*

loop values and produces the results of the LoopB node from them. The association list for a LoopB node is:

4 $\langle init \rangle \langle test \rangle \langle body \rangle \langle returns \rangle$
where $\langle init \rangle$ is the number of the subgraph corresponding to Init
$\langle test \rangle$ is the number of the subgraph corresponding to Test
$\langle body \rangle$ is the number of the subgraph corresponding to Body
$\langle returns \rangle$ is the number of the subgraph corresponding to Returns

The implicit dependencies in a LoopB node are given below:

1. All inputs to the LoopB node are available to each subgraph on the class K ports.
2. The loop values from the output of the Init subgraph are connected to the inputs of the Test, Body and Returns subgraphs on the class L ports.
3. The loop values from the output of the Body subgraph are connected to the inputs of the Test, Body and Returns subgraphs on the class L ports.
4. The results of the Returns subgraphs are connected to the output ports of the LoopB node on the class R ports.
5. When the result of the Test subgraph is false, the LoopB node makes its results available at its output ports.

Tables 3.7 and 3.8 give the port assignments and port usage in a LoopB node.

Node/ Subgraph	Input Ports	Output Ports
LoopB	K	R
Init	K	L
Test	K, L	B
Body	K, L	K, L
Returns	K, L	R

Table 3.8: *Port usage for each subgraph in LoopB*

3.7.5 Forall Node

The Forall nodes is used to express independent (as opposed to iterative) execution. The product form of **for** expressions in SISAL are translated into Forall nodes. The dot product form of **for** expressions are translated into a single Forall node whereas the cross product form requires nested Forall nodes. The Forall node has three subgraphs: *Generator, Body* and *Returns.* The Forall node may be viewed as executing independently multiple instances of an expression. The Body subgraph contains the expression to be evaluated. The Generator subgraph produces input values for each instance of the body. The Generator produces at least one multiple value on class M ports. Each element of a multiple value is sent to distinct instances of the body. A multiple value as a sequence of values, of which each value is sent to exactly one instance of the Body subgraph. The type of a multiple value is Multiple[T], where T is the type of individual values. The Returns subgraph collects the values produced by the Generator and Body subgraphs and produces the results of the Forall node. The association list for a Forall node is:

3 ⟨*Gener*⟩⟨*Body*⟩⟨*Returns*⟩
where ⟨*Gener*⟩ is the number of the subgraph corresponding to Generator
⟨*Body*⟩ is the number of the subgraph corresponding to Body
⟨*Returns*⟩ is the number of the subgraph corresponding to Returns

The implicit dependencies in a Forall node are:

1. All inputs to the Forall node are available to each subgraph on the class K ports.

2. The Body subgraph receives one value from each class M port.

Class	Usage	Port Range
K	Forall input values	$1 \ldots n_K$
M	Multiple values	$n_K + 1 \ldots n_K + n_M$
T	Results of each Body	$n_K + n_M + 1 \ldots n_K + n_M + n_T$
R	Forall results	$1 \ldots n_R$

Table 3.9: *Port assignments for each class in Forall*

Node/ Subgraph	Input Ports	Output Ports
Forall	K	R
Generate	K	M
Body	K, M	T
Returns	K, M, T	R

Table 3.10: *Port usage for each subgraph in Forall*

3. The results of the Body subgraph are available at the class T input ports of the Returns subgraph.

4. The results of the Returns subgraph are available at the class R output ports of the Forall node

Table 3.9 gives the port number ranges assigned to each class in a Forall node. Table 3.10 gives the classes of input and output ports of the Forall node itself and each of its subgraphs.

3.7.6 Select Node

The Select node is used to implement "case"-like expressions. It contains one subgraph for the selector expression and a subgraph for each alternative in the case expression. The value returned by the selector subgraph is an integer in the range $0 \ldots N-1$, which is used to select one of the N alternative subgraphs. In SISAL, there is only two-way selection. Hence all our Select nodes will contain exactly three subgraphs: the first one for the condition, the second for the true part and the third for the false part of an **if/then/else** expression. Note that **elseif** clauses result in nested Select Nodes.

In general, the association list of a Select node has the following form:

Class	Usage	Port Range
K	Select node input values	$1..n_K$
R	Select node output values	$1..n_R$
S	Selector value	1..1

Table 3.11: *Port assignments for each class in Select*

Node/ Subgraph	Input Ports	Output Ports
Select	K	R
Selector	K	S
Alternative	K	R

Table 3.12: *Port usage for each subgraph in Select*

$n\ g_0 \cdots g_{n-2}$
where n is the number of subgraphs
each g_i is an integer in the range $0 \ldots n-2$ giving the
subgraph number to use for selector value i

In our case, $n = 3$, subgraph g_0 corresponds to the false part and subgraph g_1 corresponds to the true part. Here we are treating a false value returned by the **if** condition as 0 and true value by 1.

The implicit dependencies in a Select node are:

1. All input values to the Select node are passed to the class K ports of each subgraph.

2. The result ports of each Alternative subgraph are connected to the corresponding output ports (class R) of the Select node.

Table 3.11 indicates the port ranges associated with each port class in a Select node. Table 3.12 indicates the input and output port classes for the Select node as well as for each subgraph in it.

Class	Usage	Port Range
U	Union value	1...1
V	Variant value	1...1
K	Other TagCase input values	$2 \ldots n_K$
R	TagCase output values	$1 \ldots n_R$

Table 3.13: *Port assignments for each class in TagCase*

3.7.7 TagCase Node

The **tagcase** expression of SISAL translates into a TagCase node in IF1. The TagCase node contains as many subgraphs as the number of arms in the **tagcase** expression. Several tags may share the same subgraph. The mapping of tags to subgraphs is given by the association list. Each tag is associated with a unique integer ≥ 0 based on the ordering of tags in the union type specification. The association list defines a mapping between this integer and the subgraph number for the tag. The association list takes the following form:

$n\ g_0 \ldots g_{n-1}$
where n is the number of subgraphs
g_i is the subgraph number for tag i
(g_i's need not be distinct since sharing of subgraphs is allowed)

The implicit dependencies present in a TagCase node are the following:

1. The input ports of the TagCase node are connected to the class K ports of each subgraph.
2. The value selected by the tag of the union value on port one of the TagCase node is passed to the appropriate subgraph on port one.
3. The output ports of each subgraph are connected to the corresponding output ports (class R) of the TagCase node.

Table 3.13 gives the port assignments and Table 3.14 gives the port usage for TagCase.

Node/ Subgraph	Input Ports	Output Ports
TagCase	U, K	R
subgraphs	V,K	R

Table 3.14: *Port usage for each subgraph in TagCase*

3.8 Summary

IF1 cleanly separates the frontend of the compiler from the machine specific backend. The hierarchical structure of IF1 graphs closely resembles the structure of input SISAL program. The simple and compound nodes reflect the choice of program constructs made by the programmer. The data dependencies are clearly stated either by explicit edges or by the implicit dependence rules in compound nodes. The input values to a loop type compound node are grouped together into contiguous ports so that loop invariants can be easily detected. All this makes optimization analysis on IF1 simpler. With this discussion of IF1 and that of SISAL in the previous chapter, we are now ready to describe the SISAL to IF1 translator. The next chapter describes the translator.

Chapter 4

SISAL to IF1 Translation

In this chapter, we present the implementation details of the IF1 generator. Most of the data structures and algorithms used are discussed in detail. The chapter is admittedly abstruse at places, but one could not help it because certain details seem to be inherently complex and a better way of presenting them could not be found. In fact, this is a problem with the implementation details of almost all large programs. However, in large, we have tried to put things down as succinctly as possible.

4.1 Lexical Analyzer and Parser

We used the LEX and YACC compiler tools for lexical and syntactic analysis, respectively. Although hand-coded analyzers could be more efficient, the compiler tools were used to generate the analyzers, for ease of implementation. In the following, we mention only the unusual or noteworthy aspects of the analyzers.

Keywords are stored alphabetically in a table, all in lower case. Each time an identifier is recognized by the lexical analyzer, it is first converted to lower case and then a search is performed on the keyword table with the identifier. If a match is found, the token constant corresponding to the keyword is returned. If no match is found, the token constant **NAME** is returned after the token string is stored in a name table. Each time a literal (of type integer, real, double_real, character or character string) is recognized, the token string is stored in a literal table and the appropriate token constant is returned.

The parser was built using YACC. The parser checks for grammar conformance, builds the abstract syntax structure and recovers from syntactic errors. Error recovery is necessary so that

multiple distinct errors can be detected and appropriate error messages emitted.

4.1.1 Error Recovery

The error recovery used is that provided by YACC and consequently is somewhat crude. It is based on augmenting the grammar with error productions containing special **error** tokens. Given that an error occurs, the point at which parsing should be resumed is determined by the presence of the token **error** in the grammar. When an error occurs, the parser attempts to find a production that the parser might have been in at the point where the **error** token could most recently have been returned from the lexical analyzer. Parsing continues at this point, with the token that had caused the error being the next input token. If no rule exists, parsing terminates; otherwise the parser behaves as if it sees **error** as an input token and tries to continue parsing with the rule containing the **error** token. The parser enters an error state and remains in this state until three tokens have been parsed successfully, beginning with the token at which the error occurred. If a token is seen in the error state and it cannot be parsed, it is discarded. Error messages are not printed for errors occurring in the error state to prevent a cascade of messages from being printed before the parser has recovered from the error state.

This error recovery strategy is unsophisticated; it may spawn spurious error messages in the event of certain errors. Anyhow, it suffices for the purposes of the translator, inasmuch as it allows the entire input to be processed even in the presence of errors. See [Sch85] for a detailed discussion on the error recovery mechanism provided by YACC. The strategy we have employed to insert error symbols in the grammar is quite similar to the one described in [Sch85].

4.2 Type Analysis

One of the most important tasks in semantic analysis is to check whether the input program is "type-correct". In this section, we shall see how type checking is done by our translator.

4.2.1 Representation for Types

Before we discuss the algorithm for type checking, we have to see how types are represented internally. The following record written down as a **C struct** is the answer.

```
struct T_NOD {
        enum T_CODE code;
        char *name;
        struct T_NOD *param1, *param2;
        struct T_NOD *equiv;
        unsigned label;
        /* ...misc.  fields ...*/
};
/* where T_CODE is */
enum T_CODE {
     Null, Boolean, Integer, Real, DReal, Character,
     Array, Stream, Undef, Usrdef, Function, Tuple, Multiple,
     Record, Field, Union, Tag,
};
```

The above struct is used to represent the basic types, array and stream types, record and union types, fields of records, tags of unions, user-function prototypes, user-types and other IF1 types like multiple and tuple. Which of the these types does an instance of the above struct represent depends on the value of its field code. The field label is a positive integer that is used to refer to the type in the output IF1 file. The field equiv will be discussed later.

An array is represented by a struct with code set to Array and param1 pointing to the base type of the array. If the array is the right hand side of some user-type definition, then name will be set to the user-type name (this use of name is true not just for arrays but for other structured types—streams, records and unions—as well). param2 is not used for arrays (set to NULL). Streams are represented in a similar way.

A record field is represented by a struct with code set to Field. The field name contains the name of the field. param1 points to the type of the field. param2 points to the next field in the record if there is one, otherwise it is set to NULL. A record is represented by a struct whose code has the value Record and param1 pointing to its first field. param2 is not used in records. Unions and tags are represented in a manner identical to records and fields.

We have seen that in IF1, certain nodes used in LoopA, LoopB and Forall produce "multiple" values as output or take such values as input. Just like arrays, multiple values are represented

with **param1** pointing to the base type.

A user-type is represented with the field **name** storing its name and **param1** pointing to the type on the right hand side of its definition. **param2** is not used here.

An expression with arity more than one, has the type tuple. A tuple is a list of **structs** joined through the **param2** field. Each **struct** has **code** equal to **Tuple** and **param1** pointing to the type of a component expression.

A user-defined function is represented by a **struct** with **code** having the value **Function**. **param1** points to the first argument in the formal argument list of the function. If the function has no arguments, then **param1** is **NULL**. **param2** points to the first type in the return type list. Both argument list and return type list are represented as tuples. The field **name** is used for the argument names.

4.2.2 Type Checking Algorithm

We have already seen the rules for type conformance, in Chapter 2. The unification algorithm [Aho86] is used to implement them. The algorithm is capable of handling cyclic types. The idea is to build equivalence classes of types. Two types are in the same equivalence class if they conform according to the rules in Chapter 2. Each equivalence class has a representative **struct** and all other **structs** in the class point to the representative through a chain of **equiv** pointers. The **equiv** field of the representative is set to **NULL**. Given a type it is easy to find the representative—just follow the **equiv** pointers until a **struct** with **equiv** equal to **NULL** is reached.

The type checking algorithm takes as input two types **typ1** and **typ2** and returns 1 or 0 according as the types conform or do not conform. Figure 4.1 gives the algorithm using a C-like syntax.

4.3 An Overview of the Translation

Translation of types into IF1 form, from our representation for types is straightforward. We need not output all the type **structs**—only the representative type from each equivalence class is output. All types in an IF1 file have unique labels irrespective of their scope of definition in the

```
Algorithm: unify;
Inputs:
  two types typ1 and typ2;
Output:
  1 if typ1 and typ2 conform; else 0;
{
  m = equiv_class_rep(typ1); /* the equivalence class */
  n = equiv_class_rep(typ2); /* representative        */
  eq = 1;
  if (m == n) return 1;
  n->equiv = m;
  if (m->code == n->code)
    switch (m->code) {
      case Array,Multiple,Record,Stream,Usrdef:
        eq = unify(m->param1,n->param1);
        break;
      case Function,Field,Tag:
        eq = strcmp(m->name,n->name) == 0;
      case Tuple:
        if (eq) eq = eq && unify(m->param1,n->param1);
        if (eq) eq = eq && unify(m->param2,n->param2);
        break;
      default:
        eq = 1;
    }
  else if (m->code == Usrdef)
    eq = unify(m->param1,n);
  else if (n->code == Usrdef)
    eq = unify(m,n->param1);
  return eq;
}
```

Figure 4.1: *Type Checking Algorithm*

SISAL program. For convenience, all IF1 type descriptors are written into a file separate from that of graphs (see Appendix D).

A SISAL compilation unit contains a set of function definitions. Equivalently, an IF1 file contains a set of graphs denoting functions. The body of a SISAL function is an expression of arity equal to number of types specified in the return type list of the function header. The equivalent IF1 graph body contains a set of nodes (simple and/or compound) and edges from the graph input ports to the nodes, edges connecting the nodes and edges from the nodes to the graph output ports. Each input port of the graph corresponds to a parameter of the SISAL function. Similarly, each output port of the graph corresponds to a return value of the SISAL function.

Our translator goes about translating a SISAL function definition into an IF1 graph as follows. If the function is a **global** function, an imported function descriptor (which is nothing but an IF1 line containing the letter `I`, the label of the type denoting the function, and the name of the function) is generated. If the function appears in the **defines** clause, then an exported function descriptor is generated (the letter `X` is used instead of the `I` above). If the function is local to the compilation-unit the letter `G` is used in the descriptor.

For **global** functions, the IF1 translation contains just the descriptor. Other functions have the descriptor followed by the function body. The parameters of the function are available at ports on the function's graph boundary. The return values are passed to the output ports on the graph boundary. Recall that the graph boundary is treated as a node with label 0. A node-port pair such as $(0, n)$ in an edge denotes an input port if it is used as the source and an output port if it is used as a destination.

The function body is translated into a set of nodes and edges. A question we should ask here is whether we can output code into the IF1 file, as and when a SISAL expression is parsed by the parser. It is difficult because the actual port number of edges in compound nodes will not be known until the end of the corresponding SISAL expression is reached. For example, the port number of the first loop-name in a LoopA node is one greater than the number of non-local names used in the loop. The number of non-local names will be known only after the last return clause in the loop is parsed. Therefore, we defer the output of IF1 code into a file until the function body is completely parsed. Until then, we store the IF1 code in a suitable format (to be

described below) at a convenient place.

The code is stored as a linked list whose nodes have the following type.

```
struct I_CODE {
        char type; /* one of [ 'G','I','X','N','E','{','}' ] */
        void *attrib;
};
```

If type is 'G', 'I' or 'X', the attrib field points to a T_NOD denoting the type of the function. If it is 'N', 'E', '{' or '}', the attrib field points to a representation of a simple node, an edge, a compound node opening or a compound node closing.

4.4 Representation for IF1 Nodes, Edges etc.

The various IF1 entities are represented in our translator as described in the following sections.

4.4.1 Simple Nodes

The C struct below depicts a simple IF1 node.

```
struct SIMPLE_NODE {
        IF1_NODE node;
        int label;
};
/* IF1_NODE is an enum of IF1 simple nodes (Appendix B) */
```

The field node gives the type of the node (eg. Plus, Minus...). The field label gives the label of the node as used in edges connecting output ports of the node to ports of other nodes or graph boundary.

4.4.2 Edges

Edges connect input ports of a graph to nodes, output ports of nodes to input ports of nodes, and output ports of nodes to output ports of graphs. Edges may also be literal edges that have no source. Literal edges send constant values to input ports of nodes and output ports of graphs. If we view literal edges as originating from invisible nodes (henceforth called *literal nodes and ports*), all edges may be considered as connecting two ports. In other words, an edge is just a pair of ports. A port is described by the label of the node it belongs to, the port class, and a positive integer indicating its position on the node relative to the first port in the class (henceforth called *relative port number*). The label of the node is zero if the port lies on a graph boundary or on a literal node. The port class is a character that indicates which class (K, L, M, T, etc.) the port belongs to. The port class is immaterial if the node label is non-zero (it is set to 'O' by default). If it is zero, it is a character denoting the class of the port. The actual port number can be easily calculated from the port class and the relative port number. If the port is a literal port, the port class is set to 'I'. In this case, the relative port number is an index to a table of literal constants. A character string corresponding to the literal constant is stored at that index in the literal table.

An edge is therefore, represented by the following `struct`.

```
struct EDGE {
        char sc; /* source port class */
        int sp; /* source (relative) port number */
        int sn; /* source node label */
        char dc; /* destination port class */
        int dp; /* destination (relative) port number */
        int dn; /* destination node label */
        struct T_NODE *tp; /* type of value carried by the edge */
};
```

4.4.3 Compound Nodes

The beginning of a compound node is marked by a { and the ending by a }. Henceforth, we use the terms *compound node opening* and *compound node closing*. The two are represented by the structs shown below:

```
struct COMP_NOD_BEG {
        IF1_NODE node; /* type of the node (LoopA, LoopB, Select...)*/
        int nK; /* number of class K ports */
        int nL; /* number of class L or class M ports */
        int nT; /* number of class T ports */
};

struct COMP_NOD_END {
        IF1_NODE node; /* type of the node (LoopA, LoopB, Select...)*/
        int *assoc; /* association list n,a1,a2,...an */
        int label; /* label of the node */
};
```

In COMP_NOD_BEG, nK, nL, nT denote the number of ports of various classes. Their values are not known at the time when the "compound node beginning" is generated. They are set to their values after the end of the compound node is reached. The same field nL is used for both class L and class M ports because only one of the two classes can occur in a compound node. Also note that number of ports of classes such as R need not be stored.

4.5 Representation for Simple Expressions

This section is concerned with the internal representation of simple expressions. In SISAL, an expression may be of arbitrary arity. A SISAL expression, in general, is represented by the following struct.

```
struct EXP {
        struct NOD_L *nl;
        struct T_NOD *tp;
};
/* where NOD_L is */
struct NOD_L {
        unsigned node;
        char class;
        unsigned port;
        struct NOD_L *nxt;
};
```

A single arity expression has a source node and port. If the source port is on a compound node subgraph boundary, it has a port class (such as 'K' for imported values, 'L' for loop values etc.). Each NOD_L stores these attributes of an expression of single arity. The EXP represents an expression of arbitrary arity. The field nl points to a linked list of NOD_Ls. The field tp points to the type of the expression. tp points to a non-tuple if the expression is of arity one; otherwise it points to a tuple, whose component types are the types of individual expressions.

4.6 Blocks

The above representation of simple expressions can be used for the list of values returned by a structured expression like **if-then-else, tagcase** and **for**. We also need a mechanism that reflects the block structure of SISAL and the hierarchical structure of IF1 graphs. In general, a function graph embodies several levels of nesting. The outermost level is the graph body. Inside the graph body there may be several compound nodes one nested within another. In SISAL there are also **let** expressions that add to the block structure of a program. **let** expressions are however translated to IF1 as a sequence of nodes and edges, without introducing any nesting.

The block table is used to reflect the above block structure. A block in the block table may contain the following:

- the type of the block viz. Let, TagCase, LoopA, LoopB, etc.

- table of local value-names, loop-names, temporary-names, index-names etc.
- table of imported value-names.
- IF1 code generated for the block (linked list of I_CODEs).
- miscellaneous data such as the number of ports of a particular class etc.

The block table is essentially a stack with elements of the type shown below.

```
struct BLOCK {
       enum BLK_TYP btype;
       void *attrib;
} /* where BLK_TYP is */
enum BLK_TYP {
     Function, Let, Select, Select2, TagCase,
     Loop, LoopA, LoopB, Forall, Forall2
};
```

btype indicates the type of the block. attrib points to a struct that is determined by the type of the block. The attrib field points to different types of structs for different types of blocks—a let block will not have a table of loop values whereas a LoopA or LoopB block will have such a table. The top of the block table corresponds to the current block (the block being currently parsed). The bottom most block will always be a block of type Function (the function body being currently parsed). We shall discuss more about blocks when we consider the problem of translating structured expressions.

4.7 Translation of Simple Expressions

The YACC input file contains a list of grammar rules with action associated with some rules. The action associated with a rule is executed when the parser reduces its right hand side to the non-terminal on the left hand side. The action builds some abstract representation of a syntactic construct, or calls routines that do type checking or generate IF1 code, etc. In the following, we discuss the action associated with some of the most important grammar rules. We shall not

write down the rules as such, but we shall describe what the parser does when it performs the reduction for a rule. While reading the following, the reader may wish to consult the earlier sections for the various `struct` definitions.

4.7.1 Constants

When a constant is recognized by the parser in the input program, a `struct` of type `EXP` (which is our representation for an expression) is created. The type of the expression is the type of the constant which is a `T_NOD` with `code` equal to `Integer` for integer-constant; for character strings a `T_NOD` with `code` set to `Array` and `param1` pointing to a `T_NOD` with `code` equal to `Character`; and so on. The `nl` field has its `node` set to zero, `class` set to `'I'` and `port` set to the index of the literal table where the constant is stored.

4.7.2 Value-Names

A value-name can occur at many places in an input program. If it occurs on the left hand side of a definition or a declaration it has to be inserted in the appropriate local symbol table of the current block. If it occurs as an index-name or as an element-name of a product form **for** expression, it is inserted in the index-name table or element-name table of the current `Forall` block.

Value-name references can be of two types: with and without the **old** modifier. A name with an **old** modifier denotes a value-name different from one without it. Hence the two must have different entries in a symbol table. Therefore, the key for symbol table search is not just a name but a pair (*oldbit*, *name*) where oldbit is a single bit indicating whether the name has an **old** modifier. If a value-name is not found in the current block, we look for it in blocks below it in the block table (i.e., the outer blocks in the input program text). The search ends with the `Function` block. If the value-name is still not found then an error is signalled. If it is found the value-name is inserted in all blocks in the block table above the block in which it is found. Before we proceed with this discussion, we have to see what are the attributes of a value-name. The following `struct` tells us.

```
struct VAL_NAME {
        char oldb;
        char *name;
        struct T_NOD *type;
        char classt;
        int portt, nodet;
        char classf;
        int portf, nodef;
};
```

oldb and name are the oldbit and the name of a value-name. type is the type of the value-name. classt, portt and nodet are the port class, relative port number and node label respectively, of the value-name. These three fields shall be referred to as the *origin* of the value-name. If the value-name was imported from a lower block in the stack, classf, portf and nodef are the port class, relative port number and node label respectively of the value-name in the lower block.

When a value-name definition is parsed, the classt, portt and nodet of each value-name are set to the class, port and node of the corresponding expression on the right hand side. When a value-name is imported into a block its classt is set to 'K', its portt is set to one plus the number of class K ports in the block, and nodet is set to zero.

The action associated with the grammar rule for a value-name reference is to construct a struct EXP with an nl whose class, port and node are the classt, portt and nodet of the value-name. The tp field of the expression is set to type of the value-name.

4.7.3 Binary Operations

$$e_1 \; op \; e_2$$

A binary operation comprises two expressions and a binary operator. Every binary operator in SISAL has a corresponding IF1 simple node. The translation of a binary operation shown above is done as follows.

1. Check if the operands have valid types.

2. Generate a simple node *op*.

3. Generate an edge from e_1 to the first port of *op*. This means that an edge with source class, port and node values equal to that of e_1 is generated. This meaning of "generating an edge from an expression" holds for the remainder of this chapter.

4. Generate an edge from e_2 to the second port of *op*.

5. Return an expression ('O', 1, label of *op*, result type of *op*). When we talk of an expression (c,p,n,t) we essentially mean a `struct EXP` with `nl` pointing to {c,p,n} and `tp` pointing to t.

4.7.4 Unary Operations

$$op\ e$$

The translation method is identical to binary operations. The only difference is, here we need to generate only one input edge instead of two.

4.7.5 User Function Calls

$$f(e_1, e_2, \ldots, e_n)$$

The translation is described below:

1. First perform a search on the type table to find the type of the function with name f. If the search fails check if it is a predefined function. Predefined functions are translated as described in the next section. If it is not even a predefined function then an error is flagged.

2. Check if the actual argument types conform with the formal argument types.

3. Generate a `Call` node (say, C).

4. Generate a literal edge from function-name f to the first input of C.

5. For $1 \leq i \leq n$, generate an edge from e_i to the $(i+1)^{th}$ port of C.

6. Let m be the number of return values of the function. Return an expression whose `nl` points to a list of m `struct NOD_Ls` whose `class` fields are 'O', `port` fields are progressively set to $1 \ldots m$, `node` fields are set to the label of C. The `tp` field of the expression is set to the return type of expression.

4.7.6 Predefined Function Calls

Each predefined function in SISAL has an equivalent IF1 simple node. The translation consists of three steps namely argument type conformance checking, generation of the simple node, and generation of input edges of the node.

4.7.7 Array Operations

Array Reference

An array reference is of the form:

$$a[e_1, e_2 \ldots e_n]$$

The IF1 translation is composed of the following steps.

1. Check if a is an array with at least n dimensions and if all e_is are integers.
2. Generate n `AElement` nodes (say, $AE_1 \ldots AE_n$).
3. Generate an edge from a to the first input port of AE_1 and an edge from e_1 to the second input port.
4. For $1 < i \leq n$, generate an edge from AE_{i-1} to the first input of AE_i and an edge from e_i to the second input.
5. Return the expression ('O', 1, label of AE_n,type of n^{th} dimension of a).

Array Generator

There are three cases:

Case I

$$array\ T[\,]$$

The empty array is translated as follows:

1. Check if T is an array type.
2. Generate an `ABuild` node (say AB).
3. Generate a literal edge from "1" to the first port of AB.
4. Return the expression ('O', 1, label of AB, T).

Case II

$$array\ T[e_1, e_2 \ldots e_m : e'_1, e'_2 \ldots e'_n]$$

Here the type T is optional. The translation is given below:

1. Check that all e_is are integers and all e'_js are of the same type. If T is present then ensure that it is an array with at least m dimensions and the m^{th} dimension has the same type as the e'_js.
2. Generate m `ABuild` nodes (say, $AB_1 \ldots AB_m$).
3. For $1 \leq i \leq m$, generate an edge from e_i to the first input of AB_i.
4. For $1 \leq i < m$, generate an edge from AB_{i+1} to the second input of AB_i.
5. for $1 \leq i \leq n$, generate an edge from e'_i to the $(i+1)^{th}$ input of AB_n.
6. Return the expression ('O', 1, label of AB_1, type), where type is T, if T is present; otherwise, it is an m-dimensional array with the type of e'_js as the base-type.

Case III

$$a[e_1, e_2 \ldots e_m : e'_1, e'_2 \ldots e'_n]$$

We assume one index-list–expression-list pair. If there are more, the following steps are to be repeated for each.

1. Check if a is an array with at least m dimensions. Also check if all e_is are integers and all e'_is have the same type as the m^{th} dimension of a.

2. Generate $m-1$ `AElement` nodes (which we shall call $AE_1 \ldots AE_{m-1}$). Also, generate m `AReplace` nodes (which we shall call $AR_1 \ldots AR_m$).

3. Generate edges from a to the first port of AE_1 and to the first port of AR_1. For $1 < i < m$, generate edges from AE_{i-1} to the first port of AE_i and to the first port of AR_i. Generate an edge from AE_{m-1} to the first port of AR_m

4. For $1 \leq i < m$, generate edges from e_i to the second port of AE_i and to the second port of AR_i. Generate an edge from e_m to the second port of AR_m.

5. For $1 \leq i \leq n$, generate an edge from e'_i to the $(i+2)^{th}$ port of AR_m.

6. Return the expression ('O', 1, label of AR_1, type of a).

4.7.8 Record Operations

Record Reference

$$r.f$$

The translation of the above record reference is as follows:

1. Check if r is a record and f is a field in it.

2. Generate an `RElements` node (say, RE).

3. Generate an edge from r to RE.

4. Return the expression ('O', p, label of RE, type of f) where p is an integer such that f is the p^{th} field of r.

4.7.9 Record Generator

There are two cases:

Case I

$$record\ T[f_1 : e_1; f_2 : e_2; \ldots f_n : e_n]$$

The type T is optional. The translation is given below:

1. Check that e_is have single arity. If T is present make sure that it is a record with fields $f_1, f_2 \ldots f_n$ in that order; for $1 \leq i \leq n$, check that e_i is of the same type as f_i.
2. Generate an `RBuild` node (say, RB).
3. For $1 \leq i \leq n$, generate an edge from e_i to the i^{th} input of RB.
4. Return the expression ('O', 1, label of RB, type) where type is T, if T is present; otherwise it is a record with fields $f_1, f_2 \ldots f_n$.

Case II

$$r\ replace[f_1.f_2 \ldots f_n : e]$$

We assume one field-name-list–expression pair. If there are more than one, repeat the following steps.

1. Check that r is a record. Check that f_1 is a field of r, f_2 is a field of $r.f_1$, f_3 is a field of $r.f_1.f_2$, and so on. e must be of the same type as $r.f_1.f_2 \ldots f_n$. In the following, assume that f_1 is the p_1^{th} field of r, f_2 is the p_2^{th} field of $r.f_1$, f_3 is the p_3^{th} field of $r.f_1.f_2$, and so on.
2. Generate $n-1$ `RElements` nodes (which we shall call $RE_1 \ldots RE_{n-1}$). Also, generate n `RReplace` nodes (which we shall call $RR_1 \ldots RR_n$).
3. Generate edges from r to the first ports of RE_1 and RR_1. For $1 < i < n$, generate edges from RE_{i-1} to the first ports of RE_i and RR_i. Generate an edge from RE_{n-1} to the first port of RR_n.

4. For $1 \leq i < n$, generate an edge from RR_{i+1} to the p_i^{th} port of RR_i. Generate an edge from e to the p_n^{th} port of RR_n.

5. Return the expression ('O', 1, label of RR_1, type of r).

4.7.10 Union Operations

Union Generator

$$union\ T[t : e]$$

The translation of union generator given above comprises the following steps:

1. Check that T is a union, t is a tag of T and the type of e and t are the same.

2. Generate an RBuild node (say, RB).

3. Let t be the p^{th} tag of T. Generate an edge from e to the p^{th} port of RB.

4. Return the expression ('O', p, label of RB, T).

Union Test

$$\text{is } t(e)$$

e is a union-type expression and t is a tag of the union. The above union test may be transformed to the **tagcase** expression shown below:

tagcase e
tag t : **true**
otherwise: false
end tagcase

The above **tagcase** expression has the following IF1 translation:

C Nodes and edges of e follow...
⋮
C let n and p be the node and port of e.

C let T be the type of e.

C let u be the type reference label of T.

C the TagCase node follows...

{

G 0

L 0 1 1 "T"

G 0

L 0 1 1 "F"

} m TagCase k a_1 $a_2 \ldots$ a_k

C m is the label of the TagCase node.

C k is the number of tags in T.

C for $1 \leq i \leq k, a_i = 0$ if t is the

C i^{th} tag of T; otherwise $a_i = 1$.

E n p m 1 u

4.8 Translation of Compound Expressions

4.8.1 Let Expression

A **let** expression has no explicit equivalent in IF1. Instead it is translated into a sequence of nodes and edges. The translation proceeds as follows.

After the keyword **let** is recognized in the input, a **let** block is opened (that is, a block of type `Let` is pushed onto the top of the block table). A `Let` block has the following attributes.

- a table of local value-names.
- a table of value-names imported from outer blocks.
- a linked-list of IF1 code generated for the block.

When the keyword pair **end let** is recognized, the block is closed (the block is popped off the block table). Before the block is closed, the value-name tables are freed and the code-list is moved to the outer block (outer in the input program but lies below in the block table). The expression returned has its field `nl` pointing to a list of `NOD_`Ls that represent individual sub-expressions of the possibly multiple-arity expression that makes up the **let** body.

4.8.2 If-then-else Expression

An **If-then-else** expression is translated into a `Select` node with three subgraphs—one for the first conditional expression, second for the then-part and third for the else-part.The elseif-parts result in nested `Select` nodes. In our translator, the **if** causes a `Select` block to be open and subsequent **elseif**'s cause `Select2` blocks to be open. After recognizing the **end if**, the `Select2` nodes and the `Select` node are closed.

A `Select` or `Select2` node has the following attributes:

- a table of imported value-names
- a linked-list of code generated for the block.
- the next available input port of the compound node.

After an **if** is read, a `Select` block is opened. Then a compound node opening ({) is generated followed by `G 0` to mark the beginning of the selector subgraph. The nodes and edges of the selector then follow. An edge is generated to join the selector expression with the subgraph boundary. On reading **then**, `G 0` is generated. It is followed by the nodes and edges for the then-part. Edges are generated to join the individual subexpressions following **then** to the subgraph boundary. When an **elseif** is read, action similar to that after reading **if** is taken; however a `Select2` block is opened instead of a `Select`. The then-part of **elseif** clause is handled exactly as in **if** clause. After reading **else**, the steps taken are identical to that in a then-part. On reading **end if**, the following course of action is taken.

1. Generate a } with appropriate label and association-list.
2. Generate edges to the compound node for importing non-local value-names.
3. Free the value-name table and move the code-list to the outer-block.
4. Close the block.
5. Repeat the above steps for each `Select2` and the `Select` node.

The expression returned as the result has its field `nl` pointing to a list of m `struct NOD_Ls`, where m is the arity of the **if-then-else** expression. The `struct NOD_Ls` have their `class` fields set to 'O', `port` fields set $1 \ldots m$, `node` fields set to the label of the `Select` node.

4.8.3 TagCase Expression

A **tagcase** expression is translated to a TagCase compound node. Each branch of the **tagcase** expression corresponds to a subgraph in the TagCase node. The association of tags to subgraphs is depicted in the association list.

When a **tagcase** expression is seen in the input program, the translator opens a TagCase block. A TagCase block has the following attributes:

- a pointer to tagcase variant.
- a pointer to tagcase selector expression
- a table of value-names imported into the block.
- linked list of code generated for the block.
- a positive integer indicating the next available input port of the compound node.

The IF1 code for the tag selector expression should appear outside the TagCase node. After recognizing the selector expression, a TagCase block is opened. The tag selector expression field of the block is set to its value. If there is a variant name, then make the variant field of the block point to a `struct VAL_NAME` with `classt, portt, nodet` set to 'O', 1 and 0 respectively; keep its type field equal to NULL. A compound node opening ({) is then generated.

On recognizing each tag list, the translator does the following.

1. Ensure that the tag-names are not repeated. Also, make sure that tags in the same tag list have the same type.
2. Set the type of variant (if there is one), to the type of tags in the current tag list. This ensures that when a reference to the variant is made in the tagcase branch expression, it (the variant) has the correct type. An otherwise clause resets the type back to NULL.
3. Generate a `G 0` to mark the beginning of a subgraph.
4. Generate code for the tagcase branch expression.
5. Join individual single arity subexpressions in the branch expression to the graph boundary.

On reading **end tagcase**, generate a compound node closing }. The generation of association list is a bit difficult. We have to keep track of each tag list. Let t_{ij} be the j^{th} tag-name of the i^{th} list. (Otherwise clauses are expanded to a list of tags not specified in other tag-lists). Let $\#t_{ij}$ denote the order in which t_{ij} appears in the union definition. Now the association list is

$$m\ a_1\ a_2 \ldots a_m$$

where m is the number of tags in the union; $a_k = i$ s.t. $\#t_{ij} = k$ for some k.

4.8.4 For Expression

Iterative Version

The iterative version of **for** loop is translated to either `LoopA` or `LoopB` depending on whether the termination test is after or before the loop body. The **for** loop results in a `LoopA` or `LoopB` block being pushed onto the top of the block table. The attributes of the block are

- table of imported value-names.
- table of loop-names (There are four copies of the table one for each subgraph. We call these tables Init loop-name table, Body loop-name table, Test loop-name table and Returns loop-name table. In the Init table, each loop-name has its port on a node inside the Init subgraph, rather than on the graph boundary. In the Body table, each loop-name has two copies one for the current value of the loop-name and the other for the old value—recall the use of `oldb` field in `struct VAL_NAME`. The Body table also contains temporary-names introduced within the body. The old loop-names have their ports on the graph boundaries. Loop-names and temporary-names have their ports on nodes inside the Body subgraph. The Test table contains loop-names (as well as old loop-names and temporary names if the loop is a LoopA); they have their ports on the graph boundary. Returns table contains only loop-names and they have their ports on the graph boundary).
- linked list of code generated for Init, Body, Test and Returns subgraphs.
- last node label in Init subgraph, in Body subgraph (We have to keep track of the labels because while translating the return expression list, code may have to be inserted in the

Init and Body subgraphs—a node inserted must have a label that is different from earlier nodes in the subgraph).

- the current status of the block (one of the five values—`Init`, `Body`, `Test`, `Returns` or `EndFor`. It tells us which part of the loop is being currently parsed. When a loop-name or temporary-name is referenced, which of the four tables must be searched is determined by the status. Similarly, in which linked-list should the code generated be inserted is also determined by the status field. When the loop status is `Returns` the code is inserted into both Init and Body lists. If the status is `EndFor`, the code is inserted into Returns list).

- next available class K port, class L port, class T port, class R port etc.

After reading the keywords **for initial**, a loop block is opened. At this stage, the translator does not know if it is a `LoopA` or a `LoopB`. So tentatively, it sets the type of the block to `Loop`. Set the status to `Init`. Then a compound node opening `{` and `G 0` are generated. The value-names introduced in the Init section are entered into the Init table. The last node label in the initialization subgraph is remembered because at a later stage, we may have to insert code into the Init subgraph.

On recognizing the keywords **while** or **until**, check if the type of the block is still `Loop`. If so—it means that the termination test precedes the iterator body—set the type to `LoopB` and copy the Init table to Test table with the origin fields in each entry changed so that the loop-names now belong to ports on the graph boundary. Otherwise, copy the Body table to Test table, with the origin fields suitably changed. Generate `G 0` to mark the beginning of the test subgraph. Also set the status to `Test`.

On recognizing the keyword **repeat**, check if the block type is still `Loop`. If so, set it to `LoopA`. In any case, copy the Init table to Body table with the old bit set and origin fields suitably changed. Set the status to `Body` and generate `G 0`.

After parsing both iterator and termination-test (which may be in either order depending on the type of loop—LoopA or LoopB), the Init table is copied to the Returns table. On reading **returns**, the status is set to Returns. While processing the return expressions and masking clauses, every node and edge generated are generated in duplicate; one to be inserted in the Init code list and the other in Body code list. A structure that represents the return expression part

is also built at the same time. We shall begin to generate the Returns subgraph only after the last return clause is parsed. Therefore we need to store the structure of returns part somewhere. The structure is a linked list in which each node is of the following type.

```
struct CLAUSE {
        char builder;
        struct EXP *ret,*mask;
};
```

Recall that a return clause has six parts: an optional old modifier, a constructor (**value, array, stream**), an optional reduction operator and its direction of associativity, an expression, and an optional masking clause expression. Four bit fields are packed into the field `builder`. The least significant bit indicates the old modifier, the next two bits are for the constructor, then three bits for reduction operator and the most significant two bits for the direction. If a bit field is not used, it is set to zero by default. The fields `ret` and `mask` are for the expression and masking clause respectively.

On reaching the end of returns part, edges connecting the nodes producing loop-names to the graph boundary, are generated in both Init and Body subgraphs. Status of the block is set to `EndFor`. Generate a `G 0` to indicate the beginning of Returns subgraph body. Generate the nodes and edges in the subgraph. Each return clause results in a separate output port for the subgraph. Old modifiers are translated to `RestValues` nodes; array/stream constructors to `AGather` nodes; value constructors to `FinalValue` (when there is no reduction operator) and to `Reduce`, `RedLeft`, `RedRight` or `RedTree` (when there is a reduction operator). After generating the Returns subgraph body, the four code list are coalesced into one. A compound node closing is then added at the end. The code list is moved to the outer block, the value-name tables are freed, the `nK`, `nL`, `nT` fields of the compound node opening are set to their values, the block is popped off. Then an expression to be returned as the value of the loop is constructed.

Parallel Version

The parallel version of **for** expression is translated to a `Forall` node. The translation is in many ways similar to the non-product form. The attributes of a `Forall` block are:

- a table of imported value-names.
- tables of element-names, index-names and temporary-names.
- linked-lists of code for Generator, Body and Returns subgraphs.
- status of the loop (`Generator, Body, Returns, EndFor`)
- next available class K port, class M port, class T port, class R port etc.

in expressions generating integers within a range are translated to `RangeGen` nodes. Array/stream generators are translated to `AScatter` nodes. A **dot** expression list is translated to a Generator subgraph with multiple `RangeGen` or `AScatter` nodes. A **cross** expression is translated to a `Forall` node with nested `Forall` nodes (Note that an array/stream generator with multiple index value-names implies a cross product). The nested blocks have the type `Forall2` just as nested blocks in `Select` have the type `Select2`.

4.9 Conclusion

We have thus come to the end of our discussion of SISAL to IF1 translation. But we have not yet addressed one important issue. How do we know that our translator produces correct code?

This is a rather difficult question. Until formal program verification techniques become commonplace, establishing program confidence will continue to be cumbersome. Our method of testing is admittedly crude. The translator was tested with a number of carefully chosen test programs. The code generated was then examined manually and found to be correct. The tests exposed a few bugs in the translator which have been removed.

The test programs were chosen in such a way that each type of SISAL data structure and expression appeared in at least one program. In particular, most of the input programs contained **for** expressions and array operations which are the most difficult to implement. We have taken special care to ensure that every part of the translator's source code has been executed at least

once. One of the problems is that we have not been able to test the translator for large programs. Most of the test programs are less than 50 lines. The problem is that even if the input program contains only a few hundred lines, the IF1 code produced is so large that testing it manually is a painful job. It is in such cases an IF1 interpreter or a program that generates a graphical representation of IF1 code could be of great help.

Testing a program in this manner is not 100% reliable. But it has revealed that the translator produces correct code for a good number of input programs.

Chapter 5

Future Work and Conclusion

This chapter concludes the thesis after outlining the various possible continuations of this project.

5.1 IF1 Interpreter

The testing of the code generated by our translator was done manually, which is an onerous and error prone job. If we could write an IF1 interpreter for the code generated, the testing job can be simplified to a great extent. This helps to bring out not only the errors in the translator, if any, but also the logical mistakes in the input SISAL program. Until a full-fledged SISAL compiler is ready, the interpreter can be put to immense use. But it should also be pointed out that the interpreter is bound to be extremely slow for computationally intensive programs.

5.2 Pictorial Representation of IF1 graphs

A pictorial representation of IF1 graphs is far more readable to a human being than the textual output produced by the IF1 generator [1]. A pictorial representation similar to the figures in Appendix C can be very useful if one wants to compare the output of the IF1 generator and that of the optimizer. A program could be written to translate IF1 into a suitable graphics language.

5.3 Optimization of IF1 graphs

The code generated by our translator is not of high quality. It produces a lot of redundant code. One should not expect the code generated by a straightforward translation of IF1 graphs to run efficiently on any machine.

[1] it is exactly the opposite for a machine

The IF1 graphs are amenable to a range of machine independent optimizations. Most of the classical optimization techniques can be easily applied to IF1 graphs [Ske85b]. The optimizations might include inline expansion of function calls, common subexpression elimination, loop invariant removal, etc. These techniques are described in [Aho86] and [Bar88]. They—as applied to IF1 graphs—are described in [Ske85b]. In imperative languages, function and procedure calls make automatic program analysis very difficult. Sharing of data by common blocks, call-by-reference parameters, and aliasing all complicate the global dataflow analysis that is inevitable for many optimizations. In SISAL, this problem is absent because side effects and aliasing are prohibited by the language definition. Optimization analysis is greatly simplified by replacing a function call by a copy of the function body. Such "inline expansion" is easy to for IF1. Code replaced inline can be subjected to common subexpression elimination and loop invariant removal.

The standard common subexpression elimination (CSE) algorithm—as applied to imperative languages—appears in [Aho86]. The basic unit of analysis for an imperative language is a *basic block* (a sequence of statements that contains no embedded branches). The optimizer first partitions the intermediate code produced by the compiler into basic blocks. Then it constructs a control-flow graph whose nodes are the basic blocks. The common subexpression elimination algorithm is inexpensive, but must be conservative as evident from the following example [Ske85b],

```
B := A[I];
A[J] := C;
D := A[I]
```

A[I] is a common subexpression only if it can be determined the I≠J. If that cannot be determined, then the optimizer must assume that I=J, and not combine the expressions. If the optimizer cannot make nice assumptions (I≠J), it must assume the worst. It should be obvious that this problem stems from allowing arbitrary assignment, side effects and aliasing, and therefore cannot occur in a single-assignment language like SISAL.

Since the SISAL compiler directly produces IF1 graphs, no time is spent partitioning the intermediate code into basic blocks. IF1 graphs are generally larger than basic blocks, so more common subexpressions might be found. Moreover, any two SISAL expressions that look identical within a scope, actually produce the same value by virtue of the single assignment rule and value orientation. This simplifies the CSE algorithm.

Loop-invariant detection and removal for imperative languages can be extremely slow. Loop detection first requires the construction of a control-flow graph; then *dominators* for each basic block within the control-flow graph have to be computed. Finally, loops are detected by finding "back edges" in the control-flow graph [Aho86]. Loop detection is much simpler in IF1; all we need to do here is search the graphs for loop nodes (LoopA, LoopB and Forall).

After detecting a loop, the inputs to each expression inside its body are inspected to see if their definitions lie outside the loop. If so, the expression is loop-invariant and can be moved outside the loop. In imperative language compilers, use-definition (u-d) chains must be computed for each variable not previously defined within a block. Global data flow analysis must be done on the entire program in order to compute u-d chains.

In IF1 loop invariant analysis is much simpler. If all input edges of a node inside the loop body correspond to the input edges of the loop, the node denotes a loop invariant operation. This process is very inexpensive.

For the details of the above optimization algorithms, the reader is referred to [Ske85b].

5.4 Conclusion

A translator for the single assignment language SISAL to an interemediate graph representation IF1 has been implemented. The translator can serve as a common module in SISAL compilers for a range of sequential and multiprocessing sytems. The code produced by the translator is a straightforward translation of the source program and hence not efficient. The code can be improved by the usual optimization techniques. These techniques are easier to implement for IF1 than for conventional languages.

Appendix A

SISAL Syntax

A.1 Lexemes

- **Reserved words:**

`array`	`at`	`boolean`	`catenate`	`character`
`cross`	`define`	`dot`	`double_real`	`else`
`elseif`	`end`	`error`	`false`	`for`
`forward`	`function`	`global`	`greatest`	`if`
`in`	`initial`	`integer`	`is`	`least`
`left`	`let`	`nil`	`null`	`of`
`old`	`otherwise`	`product`	`real`	`record`
`repeat`	`replace`	`returns`	`right`	`stream`
`sum`	`tag`	`tagcase`	`then`	`tree`
`true`	`type`	`union`	`unless`	`until`
`value`	`while`	`when`		

- **Names:** A name begins with a letter followed by zero or more digits, letters or underscores (_). The number of significant characters in a name is 32.

- **Numbers:** Integer and real constants have the usual syntax. Double_reals use the letters `d` or `D` instead of the `e` or `E` used for exponents in reals.

- **Character Strings and Constants:** SISAL uses the C syntax for character constants and strings.

- **Comments:** Comments begin with a % and extend upto the end of line.

A.2 Grammar

The grammar of SISAL is given in the extended BNF notation. The left and right hand sides of a rule are separated by the symbol ::=. Non-terminals are shown in roman font. Terminal symbols (including the reserved words) are shown in **typewriter-like** font. Some terminal symbols like name, integer-constant, real-constant etc. are shown in **sans serif** font. A string of grammar symbols that appears within a pair of brackets ([ ]) is optional. An ellipsis (...) denotes a repetition (non-empty sequence) of the grammar symbol preceding it. If the ellipsis follows an optional string (i.e., enclosed within [ ]'s), it means that the string may be repeated zero or more times.

compilation-unit ::= **define** function-name-list
[type-def-part]
[**global** function-header] ...
function-def ...

function-name-list ::= function-name [, function-name] ...

function-def ::= **forward function** function-header
| **function** function-header
[type-def-part]
[function-def] ...
expression
end function

type-def-part ::= type-def [; type-def] ...[;]

type-def ::= **type** type-name = type-spec

function-header ::= function-name ([decl-list] **returns** type-list)

decl-list ::= decl [; decl] ...[;]

type-list ::= type-spec [, type-spec] ...

type-spec ::= basic-type-spec
| compound-type-spec
| type-name

```
basic-type-spec     ::= boolean
                      | character
                      | double_real
                      | integer
                      | null
                      | real
compound-type-spec  ::= array [ type-spec ]
                      | stream [ type-spec ]
                      | record [ field-spec [ ; field-spec ] ... [ ; ] ]
                      | union [ tag-spec [ ; tag-spec ] ... [ ; ] ]
field-spec          ::= field-name [ , field-name ] ... : type-spec
tag-spec            ::= tag-name [ , tag-name ] ... [ : type-spec ]
expression          ::= simple-expression [ , simple-expression ] ...
simple-expression   ::= primary [ binary-op primary ] ...
unary-op            ::=  +  |  -  |  ~
binary-op           ::=  <  |  <=  |  >  |  >=  |  =  |  ~=
                      |  +  |  -  |  |
                      |  *  |  /  |  &
                      |  ||
primary             ::= constant
                      | value-name
                      | ( expression )
                      | invocation
                      | array-ref
                      | array-generator
                      | stream-generator
                      | record-ref
                      | record-generator
                      | union-test
                      | union-generator
```

```
                    |  error-test
                    |  prefix-operation
                    |  condtional-exp
                    |  let-in-exp
                    |  tagcase-exp
                    |  iteration-exp
                    |  old value-name
                    |  unary-op primary
invocation          ::= function-name ( [ expression ] )
array-ref           ::= primary [ expression ]
array-generator     ::= array type-name [ ]
                    |  array [ type-name ] [ expr-pair ]
                    |  primary [ expr-pair [ ; expr-pair ] ... [ ; ] ]
expr-pair           ::= expression : expression
stream-generator    ::= stream type-name [ ]
                    |  stream [ type-name ] [ expression ]
record-ref          ::= primary . field-name
record-generator    ::= record [ type-name ] [ field-def [ ; field-def ] ... [ ; ] ]
                    |  primary replace [ field : expression
                                         [ ; field : expression ] ... [ ; ] ]
field-def           ::= field-name : expression
field               ::= field-name [ . field-name ] ...
union-test          ::= is tag-name ( expression )
union-generator     ::= union type-name [ tag-name [ : expression ] ]
error-test          ::= is error ( expression )
prefix-operation    ::= prefix-name ( expression )
let-in-exp          ::= let
                        decldef-part
                        in
                        expression
```

```
                    end let
decldef-part        ::= decldef [ ; decldef ] ... [ ; ]
decldef             ::= decl
                    |   def
                    |   decl [ , decl ] ... := expression
decl                ::= value-name [ , value-name ] ... : type-spec
def                 ::= value-name [ , value-name ] ... := expression
tagcase-exp         ::= tagcase [ value-name := ] expression
                        tag-list : expression
                        [ tag-list : expression ] ...
                        [ otherwise : expression ]
                        end tagcase
tag-list            ::= tag tag-name [ , tag-name ] ...
conditional-exp     ::= if expression then expression
                        [ elseif expression then expression ] ...
                        else expression
                        end if
iteration-exp       ::= for initial
                        decldef-part
                        iterator-terminator
                        returns return-exp-list
                        end for
                    |   for in-exp-list
                        [ decldef-part ]
                        returns return-exp-list
                        end for
iterator-terminator ::= iterator termination-test
                    |   termination-test iterator
iterator            ::= repeat iterator-body
iterator-body       ::= decldef-part
```

termination-test ::= **while** expression
| **until** expression

in-exp-list ::= in-exp
| in-exp **dot** in-exp [**dot** in-exp] ...
| in-exp **cross** in-exp [**cross** in-exp] ...

in-exp ::= value-name **in** expression [**at** index-list]

index-list ::= value-name [, value-name] ...

return-exp-list ::= return-clause ...

return-clause ::= [**old**] return-exp [masking-clause]

masking-clause ::= **unless** expression
| **when** expression

return-exp ::= **value of** [[direction] reduction-op] expression
| **array of** expression
| **stream of** expression

direction ::= **left**
| **right**
| **tree**

reduction-op ::= **sum**
| **product**
| **least**
| **greatest**
| **catenate**

constant ::= **false**
| **nil**
| **true**
| integer-constant
| real-constant
| character-constant
| character-string-constant
| **error** [type-spec]

prefix-name ::= `character`
| `double_real`
| `integer`
| `real`

function-name ::= name

field-name ::= name

tag-name ::= name

type-name ::= name

value-name ::= name

Appendix B

IF1 Syntax

The IF1 grammar is given below in the extended BNF. The non-terminal symbols are shown in roman font. The terminal symbols appear in a `typewriter-like` font if they stand for themselves and in sans serif otherwise. Newline denotes the new-line character, Literal denotes any string of printable characters except the new-line, PosInteger denotes a positive integer, and Integer denotes either zero or a positive integer. A string of symbols enclosed by [ ]'s denotes an optional item. An ellipsis (...) denotes the repetition of one or more times of the symbol it follows. If the ellipsis follows an optional string it represents the repetition of the string zero or more times.

File ::= [Line Comment Newline ...] ...

Line::= C

|	N Label	Node	
|	T Label	TypeTableEntry	
|	E Source	Destination	TypeReference
|	L	Destination	TypeReference "Literal"
|	L	Destination	TypeReference `ErrorValue`
|	G		TypeReference
|	G		TypeReference "Literal"
|	I		TypeReference "Literal"
|	X		TypeReference "Literal"
|	{		
|	} Label	Node	Count AssociationList

Label ::= PosInteger

```
Source          ::= SourceNode      SourcePort
Destination     ::= DestinationNode DestinationPort
SourceNode      ::= Integer
SourcePort      ::= PosInteger
DestinationNode ::= Integer
DestinationPort ::= PosInteger
TypeReference   ::= Integer
Count           ::= Integer
AssociationList ::= Integer ...
Node ::=  ForAll  | Select  | TagCase  | LoopA  | LoopB
      |   AAddH  | AAddL   | AExtract  | ABuild | ACatenate
      |   AElement | AFill | AGather   | AIsEmpty | ALimH
      |   ALimL  | ARemH   | ARemL     | AReplace | AScatter
      |   ASetL  | ASize   | Abs       | BindArguments | Bool
      |   Call   | Char    | Div       | Double | Equal
      |   Exp    | FirstValue | FinalValue | Floor
      |   Int    | IsError | Less      | LessEqual
      |   Max    | Min     | Minus     | Nod    | Neg
      |   NoOp   | Not     | NotEqual  | Plus
      |   RangeGen | RBuild | RElements | RReplace
      |   RedLeft | RedRight | RedTree | Reduce | RestValues
      |   Single | Times   | Trunc
TypeTableEntry ::= Array    TypeReference
                 | Basic    BasicType
                 | Field    TypeReference TypeReference
                 | Function TypeReference TypeReference
                 | Multiple TypeReference
                 | Record   TypeReference
                 | Stream   TypeReference
                 | Tag      TypeReference TypeReference
```

```
        | Tuple    TypeReference TypeReference
        | Union    TypeReference
BasicType ::= Boolean
        | Character
        | Double
        | Integer
        | Null
        | Real
```

Appendix C

A Sample Program and its Translation

In this appendix we present a sample SISAL program and its IF1 equivalent. The program contains a function for multiplying two matrices. We give both the textual and the pictorial representations of IF1 graphs. The IF1 code given in this appendix contains comments that were not generated by our translator—they were manually inserted.

The Input Program

```
define multiply

type matrix = array[ array[ real ] ]

function rowcol (A : matrix returns integer,integer)

array_size(A),array_size(A[1])
end function


function multiply ( A, B : matrix returns matrix )

   let
      ar, ac := rowcol(a);
      br, bc := rowcol(a[1]);
   in
      if ac = br then
        for I in 1, ar cross J in 1, bc
            IP := for K in 1, ac
                      returns
                         value of sum A[I, K] * B[K, J]
                      end for
        returns
            array of IP
```

```
      end for
    else
      error[ matrix ]
    end if
  end let

end function
```

The IF1 Code

```
T   1       Basic       Boolean
T   2       Basic       Character
T   3       Basic       Double_Real
T   4       Basic       Integer
T   5       Basic       Real
T   6       Basic       Null
T   9       Multiple    4
T  10       Array       6
T  11       Array       10      %na="matrix"
T  12       Tuple       11   13 %na="A"
T  13       Tuple       11   0  %na="B"
T  14       Tuple       11   0
T  15       Function    12   14 %na="multiply"
T  16       Multiple    6
T  17       Multiple    10
T  18       Function    28   29
T  28       Tuple       6    29
T  29       Tuple       6    0
T  30       Tuple       11   0  %na="A"
T  31       Tuple       4    32
T  32       Tuple       4    0
T  33       Function    30   31 %na="rowcol"
G           33  "rowcol"
N 1  ASize
E 0 1  1 1  11  %na="A"
N 2  AElement
E 0 1  2 1  11  %na="A"
L      2 2  4   "1"
N 3  ASize
E 1 1  3 1  10
X           15  "multiply"
N 1  Call
L      1 1  33  "rowcol"
E 0 1  1 2  11  %na="A"
```

```
N 2  Call
L       2 1  33  "rowcol"
E 0 2   2 2  11  %na="B"
{ nK=6
G            0   Selector
N 1  Equal
E 0 2   1 1  4   %na="ac"
E 0 4   1 2  4   %na="br"
N 2  Int
E 1 1   2 1  1
E 2 1   0 1  4
G            0   True Part
{ nK=5 nM=1 nT=1
G            0   Generator
N 1  RangeGen
L       1 1  4   "1"
E 0 1   1 2  4   %na="ar"
E 1 1   0 6  9   %na="I"
G            0   Body
{ nK=5 nM=1 nT=1
G            0   Generator
N 1  RangeGen
L       1 1  4   "1"
E 0 3   1 2  4   %na="bc"
E 1 1   0 6  9   %na="J"
G            0   Body
{ nK=5 nM=1 nT=1
G            0   Generator
N 1  RangeGen
L       1 1  4   "1"
E 0 1   1 2  4   %na="ac"
E 1 1   0 6  9   %na="K"
G            0   Body
N 1  AElement
E 0 2   1 1  11  %na="A"
E 0 4   1 2  4   %na="I"
N 2  AElement
E 1 1   2 1  10
E 0 6   2 2  4   %na="K"
N 3  AElement
E 0 3   2 1  11  %na="B"
E 0 6   2 2  4   %na="K"
N 4  AElement
```

```
E 3 1  4 1  10
E 0 5  4 2  4   %na="J"
N 5  Times
E 2 1  5 1  6
E 4 1  5 2  6
E 5 1  0 7  6
G           0   Returns
N 1  Reduce
L      1 1  18  "sum"
L      1 2  6   "0.0"
E 0 7  1 3  16
E 1 1  0 1  6
} 1  Forall 3 0 1 2
E 0 1  1 1  4   %na="ac"
E 0 2  1 2  11  %na="A"
E 0 4  1 3  11  %na="B"
E 0 5  1 4  4   %na="I"
E 0 6  1 5  4   %na="J"
E 1 1  0 7  6
G           0   Returns
N 1  AGather
L      1 1  4   "1"
E 0 7  1 2  16
E 1 1  0 1  10
} 1  Forall 3 0 1 2
E 0 2  1 1  4   %na="ac"
E 0 3  1 2  11  %na="A"
E 0 4  1 3  4   %na="bc"
E 0 5  1 4  11  %na="B"
E 0 6  1 5  4   %na="I"
E 1 1  0 7  10
G           0   Returns
N 1  AGather
L      1 1  4   "1"
E 0 7  1 2  17
E 1 1  0 1  11
} 1  Forall 3 0 1 2
E 0 1  1 1  4   %na="ar"
E 0 2  1 2  4   %na="ac"
E 0 3  1 3  11  %na="A"
E 0 5  1 4  4   %na="bc"
E 0 6  1 5  11  %na="B"
E 1 1  0 1  11
```

```
G            0   False Part
L       0 1  11  "error"
} 3  Select 3 0 2 1
E 1 1   3 1  4   %na="ar"
E 1 2   3 2  4   %na="ac"
E 0 1   3 3  11  %na="A"
E 2 1   3 4  4   %na="br"
E 2 2   3 5  4   %na="bc"
E 0 2   3 6  11  %na="B"
E 3 1   0 1  11
```

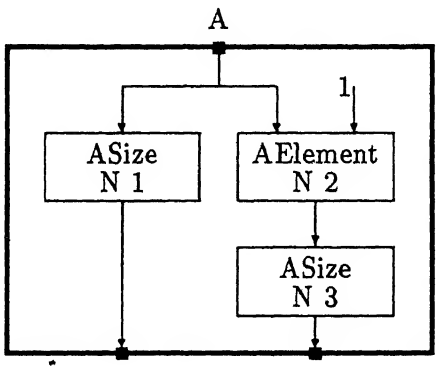


Figure C.1: *Graph of rowcol*

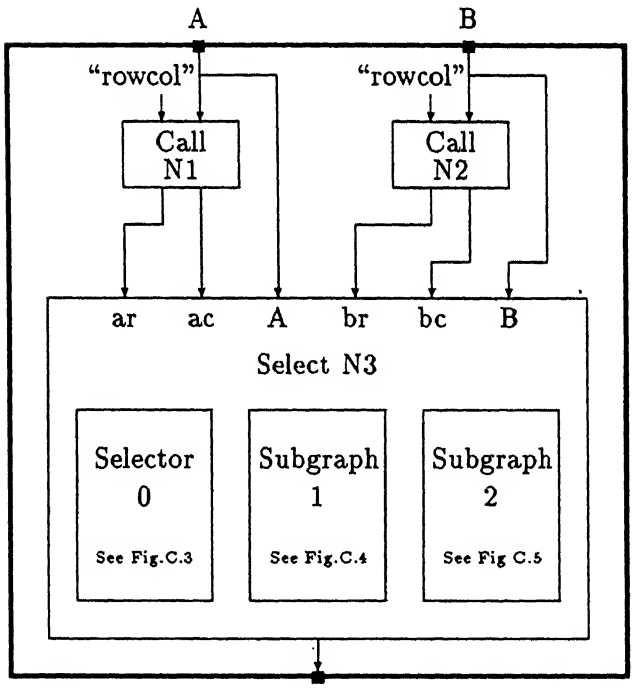


Figure C.2: *Graph of multiply*

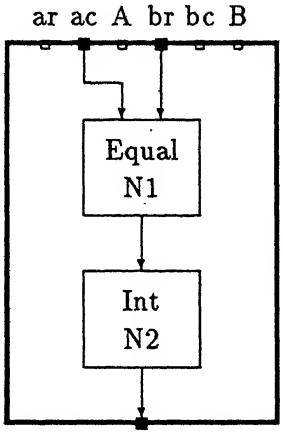


Figure C.3: *Selector*

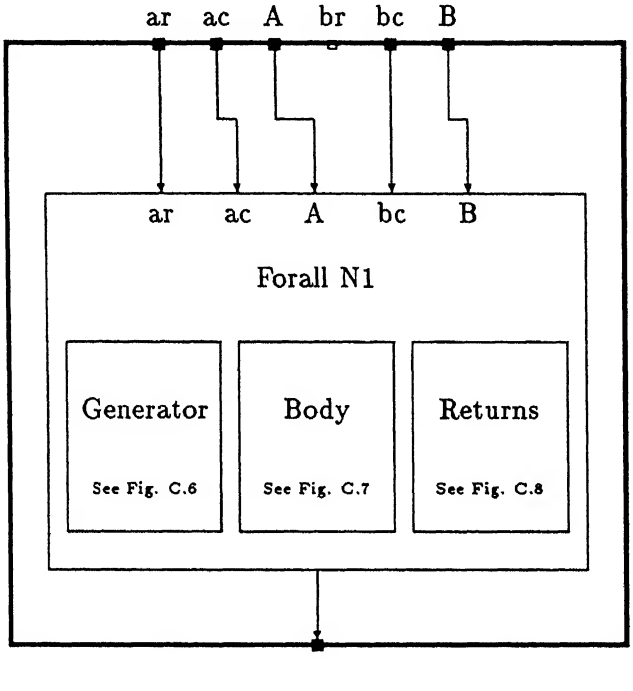


Figure C.4: *Subgraph 1*

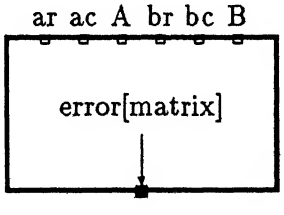


Figure C.5: *Subgraph 2*

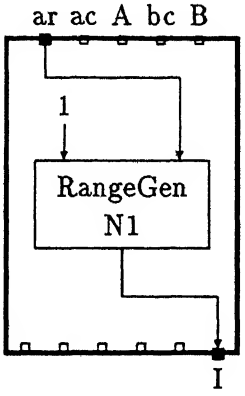


Figure C.6: *Generator 1*

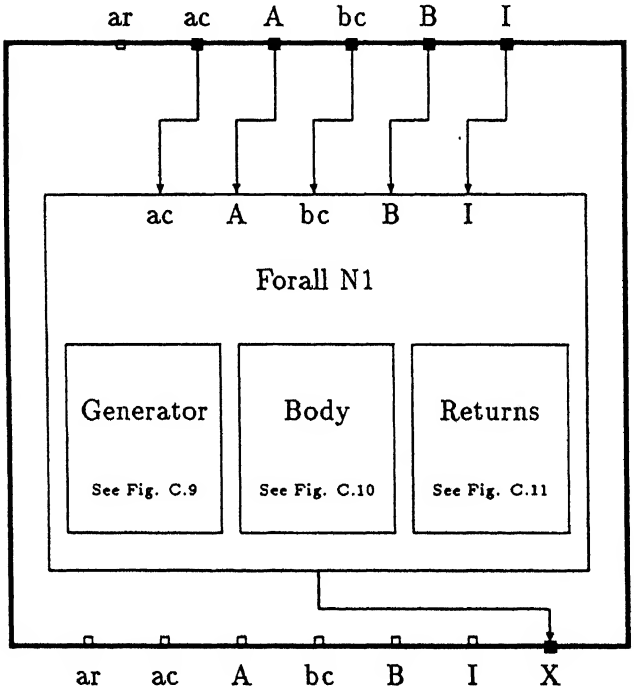


Figure C.7: *Body 1*

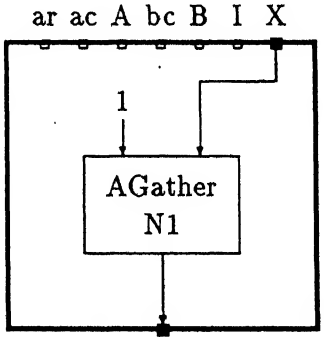


Figure C.8: *Returns 1*

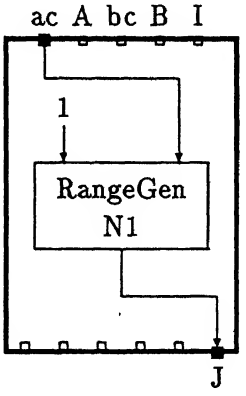


Figure C.9: *Generator 2*

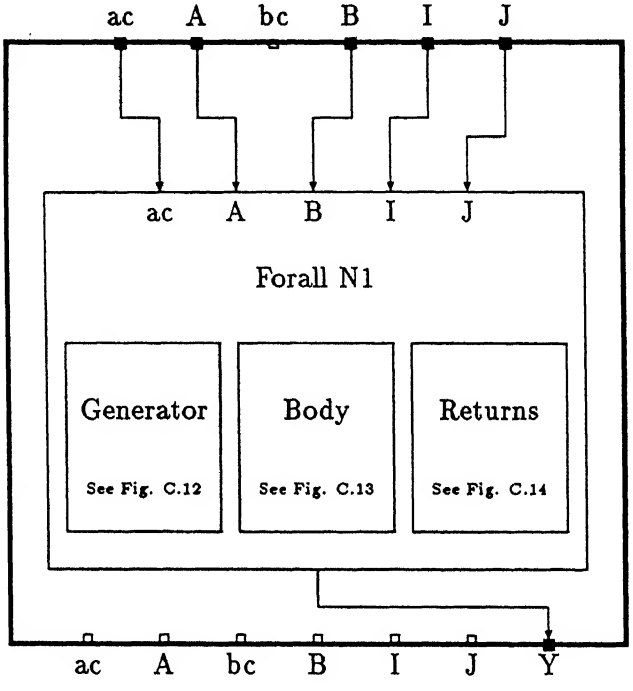


Figure C.10: *Body 2*

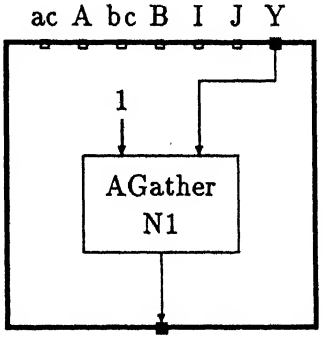


Figure C.11: *Returns 2*

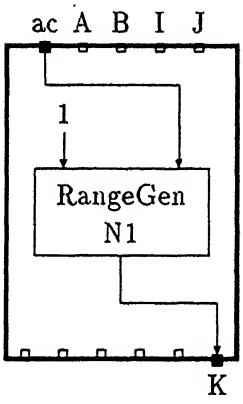


Figure C.12: *Generator 3*

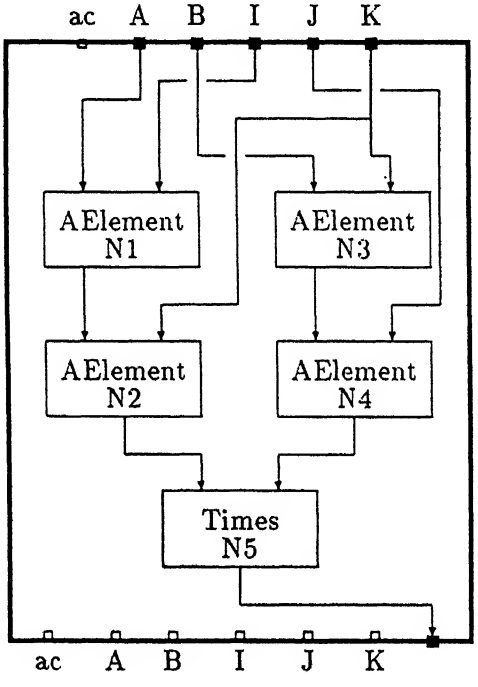


Figure C.13: *Body 3*

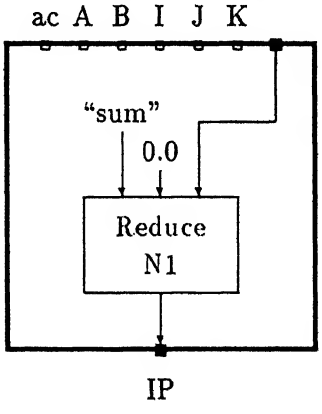


Figure C.14: *Returns 3*

Appendix D

IF1 Generator: User's Manual

This appendix is for the users of SISAL to IF1 translator. In this appendix, we shall refer to the IF1 generator as `ifg`. The first section suggests how to run the `ifg` program and describes the output files it produces. The second section describes error recovery in `ifg` and the error messages it produces. In the error messages given here, `typewriter-like` font is used for the actual computer output and *italics* for symbols that will be replaced by other strings in the actual output (eg., *name* will be replaced by an input program identifier).

D.1 The Program ifg

NAME

`ifg` - SISAL to IF1 translator

SYNOPSIS

`ifg` *file*

DESCRIPTION

The `ifg` program receives a SISAL program as input and produces its IF1 representation as output. Two output files are produced by the program. If the input file is called *file*, the output files are named *file.t* and *file.g*. The output file *file.t* contains the IF1 representation of types in the SISAL program. The file *file.g* contains IF1 graphs.

RETURN VALUE

>0 error in the input program
0 successful completion

A Sample Run

Shown below, are a small program (with errors) and the messages produced by ifg when given that program as input.

An Input Program:

```
define func1,func2
type vector = array[integer];

function func1(a,b: integer returns vector)
a + b
end function

function func2(a integer returns real
end function
```

Messages emitted by ifg for the above program:

```
[L 6] clash between expression type and return type of function "func1"
[L 8] at or before key word "integer": expecting: ',' ':'
```

D.2 ifg Error Messages

ifg does not stop at the first error in the input SISAL program. It recovers from errors and tries to find as many of them as possible using a simple error recovery mechanism. A syntactic error in the source file causes ifg to abort further semantic processing. Once a syntactic error is detected, the source file is scanned further to detect only syntactic errors. Code generation is aborted in the event of both syntactic and semantic errors. The various syntactic error messages are shown below.

at or before name: "*name*": expecting: *list-of-symbols*
at or before keyword: "*keyword*": expecting: *list-of-symbols*
at or before symbol: "*symbol*": expecting: *list-of-symbols*
at or before name: syntax error
at or before keyword: syntax error
at or before symbol: syntax error
character constant exceeds line
character string exceeds line

In the above messages, *name*, *keyword* and *symbol* denote a name, a SISAL reserved word and any other symbol respectively at or before which the error occurred. The *list-of-symbols* is a set of symbols of which one could have possibly averted the error. That is, if a symbol from the list appeared before or instead of the one that is currently being scanned, the error might not have (possibly) occurred. Some times the *list-of-symbols* will not be displayed. This is when there are too many symbols in the list—listing all of them serves no purpose other than cluttering up the display. In this case, `ifg` simply says `syntax error`.

Now we turn to semantic errors. Once a semantic error has occurred (which might be due to a type clash, an undefined name, etc.), `ifg` displays an appropriate message, aborts IF1 code generation and proceeds to find more errors in the program. The complete list of semantic error messages follows:

`redefinition of typename "`*name***`"`**
The type-name *name* is defined more than once in a type definition part.

`typename "`*name***`" unknown`**
The type-name *name* is not defined.

`type name undefined "`*name***`"`**
All type-names introduced in a type definition part must be defined exactly once. This error occurs immediately after the end of a type definition part, whereas the previous error occurs inside a function body (i.e., in an expression).

`recursive type "`*name***`" non-terminating`**
The mutual recursion in the type definition part does not terminate.

`function name "`*name***`" duplicated in DEFINE`**
Function-name *name* appears more than once in the **define** clause.

`function name "`*name***`" appears in both DEFINE and GLOBAL`**
A function-name cannot appear in both clauses.

`illegal forward (re)declaration of function "`*name***`"`**
One of the following happened:

- A forward declaration of function *name* had already appeared in the same function scope or an

outer scope.

- A definition of function *name* had already appeared in the same function scope or an outer scope.

type clash between function "*name*" and its forward declaration

The headers in the forward declaration and the definition of function *name* do not match.

forward declared function "*name*" not defined

A forward declared function must be defined before the end of its scope.

redefinition of function "*name*"

A definition of function *name* has already appeared in the same function scope or an outer scope.

argument name "*name*" duplicated in function header

The argument name *name* occurs more than once in the function header.

clash between expression type and return type of function "*name*"

The type of the expression forming the function body and the return type of the function (as it appears in the function header) do not match. This could also be a mismatch in arity. *name* is the name of the function.

type clash between actual and formal arguments of function "*name*"

The type of actual parameters in a call to function *name* do not match with the type of formal parameters. This message is also emitted when there are too many or too less actual parameters.

unknown function "*name*"

The function *name* is undefined.

definition of export function "*name*" missing

A function appearing in the **define** clause must be defined in the compilation-unit.

illegal binary operation (*type1 op type2* !)

The operand types *type1* and *type2* are illegal for the binary operation *op*.

illegal unary operation (*op type* !)

The operand type *type* is illegal for the unary operation *op*.

illegal prefix operation *op* **on** *type*

The operand type *type* is illegal for the prefix operation *op*.

too many arguments for error test

error test can have only one argument.

non-integer (*type* **!) index in array reference**

The index in an array reference must be an integer and not *type*.

non-array operand (*type* **!) in array reference**

The array reference operand is found to be of type *type* when it must be an array.

non-integer index (*type* **!) in array generator**

The index in an array generator must be an integer and not *type*.

non-array operand (*type* **!)in array generator**

The array generator operand is found to be of type *type* when it must be an array. Or, the type-name specified in the array generator is not an array.

type clash (*type1* **vs** *type2***) between elements in array generator**

All elements in the element-list for a particular dimension in an array generator must be of the same type. A type clash between *type1* and *type2* occurred.

non-stream operand (*type* **!) in stream generator**

The type-name specified in the stream generator must be a stream and not *type*.

type clash (*type1* **vs** *type2***) between elements in stream generator**

All elements in the element-list of a stream generator must be of the same type. A type clash between *type1* and *type2* occurred.

field name "*name***" duplicated in record definition**

The field-name *name* occurs more than once in the record definition

non-record operand (*type* !) in record reference

A record reference must have a record operand and not *type*.

unknown field name "*name*" in record reference

The field-name *name* does not appear in the definition of the record.

non-record (*type* !) operand in record generator

A record generator must have a record operand or a record type-name and not *type*.

unknown field name "*name*" in record generator

The field-name *name* appearing in the record generator does not occur in the definition of the record type-name.

field name "*name*" reassigned in record generator

The field-name *name* is assigned a value more than once in the record generator.

multiple arity expression assigned to field name "*name*"

A field-name in a record generator can be assigned only a single arity expression.

type clash (*type1* vs *type2*) in assignment to field name "*name*"

The type *type2* of the expression assigned to field-name *name* in the record generator clashes with *type2*, the field-name's type in the definition of the record.

tag name "*name*" duplicated in union definition

The tag-name *name* occurs more than once in the union definition.

non-union (*type* !) operand in union test

The type of expression appearing in the union test is *type* when it must be a union.

unknown tag name "*name*" in union test

The tag-name appearing in the union test does not occur in the definition of the union.

non-union (*type* !) operand in union generator

The expression type in a union generator must be union and not *type*.

unknown tag name "*name*" in union generator

The tag-name *name* does not appear in the definition of the union.

type clash (*type1* vs *type2*) in assignment to tag name "*name*"

The type *type2* of the expression assigned to tag-name *name* in the union generator clashes with *type2*, the tag-name's type in the definition of the union.

non-union (*type* !) selector in tagcase expression

The selector expression in a tagcase expression must be a union and not *type*.

unknown tagname "*name*" in tagcase expression

An unknown tag-name (*name*) appears in a **tag** clause.

tag name "*name*" duplicated in tagcase expression

The tag-name *name* occurs in more than one **tag** clause in the tagcase expression.

incompatible types for tagnames in "TAG tagname-list"

All tag-names in a **tag** clause must have the same type.

redundant OTHERWISE clause in tagcase expression

The otherwise clause is redundant because all tag names in the union type are covered by the **tag** clauses.

missing OTHERWISE clause in tagcase expression

An otherwise clause is required because all tag names in the union type are not covered by the **tag** clauses.

type clash between branches of tagcase expression

The branches of **tagcase** expression differ in type/arity.

type clash between branches of conditional expression

The branches of **if-then-else** expression differ in type/arity.

non-Boolean (*type* !) expression after *keyword*

The type of expression appearing after *keyword* (**if**, **elseif**, **while**, **until**, **when** or **unless**) is *type* when it must be boolean.

valuename "*name*" redeclared

The value-name *name* is declared more than once in the same scope. Or, the value-name has already been introduced in the same scope by a reference or a definition.

valuename "*name*" referenced but not defined

The value-name *name* is used in an expression but was not defined before.

valuename "old *name*" referenced but not defined

One of the following situations caused the error.

- A value-name which is not a loop-name is referenced with the old modifier.
- The old value of a loop-name is referenced in the initialization section or the returns section of a **for** expression.
- The old value of a loop-name is referenced in the termination test of a LoopA type **for** expression.

valuename "*name*" reassigned

A violation of single assignment rule. The value-name *name* is assigned a value more than once in the same scope.

unbalanced valuename definition (*m* valuename(s) := *n* expression(s) !)

The number of value-names on the left hand side of := does not match with the arity of expression on the right hand side.

type clash (*type1* vs *type2* !) in assignment to valuename "*name*"

The type (*type2*) of expression assigned to value-name *name* clashed with the type (*type1*) given to it by an earlier declaration.

loopname "*name*" not defined in for expression initialization

The loop-name *name* was introduced by a declaration but not defined in the initialization section of **for** expression.

element name "*name*" duplicated in generator

name occurs more than once in the **in** expression list of **for** expression.

index name "*name*" duplicated in generator

name occurs more than once in the **in** expression list of **for** expression.

index name list in integer range generator

No index-name can appear in integer range generator of a **for** loop **in** expression list.

illegal type for generator expression

The type of an **in** expression in a **for** loop generator must be one of the types `Array`, `Stream` or `(Integer,Integer)`.

DOT and CROSS products intermixed in generator

Dot and **cross** products cannot be intermixed. Note that more than one index-name for an array/stream element generator implies a **cross** product.

multiple arity expression after return clause prefix

Only a single arity expression can follow the return clause prefix in a **for** expression.

illegal reduction operation (*op* of *type* !)

The type of the expression appearing in a return clause of a **for** expression is illegal for the reduction operator *op*.

Bibliography

[Aho86] A. Aho, R. Sethi and J. D. Ullman, *"Compilers: Principles, Techniques and Tools"*, Addison Wesley, 1986.

[Bar88] W. A. Barrett, R. M. Bates, D. A. Gustafson and J. D. Couch, *"Compiler Construction: Theory and Practice"*, Science Research Associates Inc., 1988.

[McG82] J. R. McGraw, *"The VAL Language: Description and Analysis"*, *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, pp. 44–82.

[McG85] J. R. McGraw, S. Skedzielewski, S. Allan, R. R. Oldehoeft, J. Glauert, C. C. Kirkham, B. Noyce and R. Thomas, *"SISAL: Streams and Iteration in a Single Assignment Language"*, Language Reference Manual, Version 1.2, Tech. Report of CRG, LLNL, March 1, 1985.

[Sch85] A. Schriener and H. G. Friedman, *Introduction to Compiler Construction with Unix*, Englewood Cliffs, Prentice Hall, 1985.

[Ske85a] S. Skedzielewski and J. Glauert, *"IF1: An Intermediate Form for Applicative Languages"*, Language Reference Manual, Tech. Report of CRG, LLNL, July 31, 1985.

The following paper discusses some of the classical code optimization techniques as applied to IF1 graphs.

[Ske85b] S. Skedzielewski and M. L. Welcome, *"Dataflow Graph Optimization in IF1"*, *Functional Programming Languages and Computer Architecture*, Nancy (Sept. 1985), in Lecture Notes in Computer Science, Vol. 201, pp. 17–34, J. P. Jouannaud (ed.).

The following are relevant literature not cited in the body of this thesis.

The paper below describes a range of optimization techniques in a SISAL compiler for the Manchester dataflow machine.

[Böh89a] A. P. W. Böhm and J. Sargeant, *"Code Optimization for Tagged-Token Dataflow Machines"*, *IEEE Transactions on Computers* (Jan. 1989), Vol. 38, No. 1, pp. 4–14.

The following paper discusses a SISAL compiler for the Warp Systolic Array.

[Gro87] T. Gross, A. Sussman, *"Mapping a Single Assignment Language onto the Warp Systolic Array"*, *Functional Programming Languages and Computer Architecture*, Portland, Oregon (Sept. 1987), in Lecture Notes in Computer Science, Vol. 274, pp. 347–363, G. Kahn (ed.).

The following article addresses the issues in implementing a SISAL compiler for the Sequent Balance multiprocessor system.

[Old88] R. R. Oldehoeft and D. C. Cann, "*Applicative Parallelism on a Shared Memory Multiprocessor*", *IEEE Software* (Jan. 1988), pp. 62–70.

The next paper addresses the problem of automatic partitioning of SISAL programs.

[Sar88] V. Sarkar, S. Skedzielewski and P. Miller, "*An Automatically Partitioning Compiler for SISAL*", **CONPAR88**, pp. 376–383, C. R. Jesshope and K. D. Reinartz (eds.), British Computer Society Press.

The paper below presents an Id (a non-strict, single assignment, functional language) compiler for a multithreaded architecture.

[Tra91] K. R. Traub, "*Multithreaded Code Generation for Dataflow Architectures from Non-Strict Programs*", *Functional Programming Languages and Computer Architecture*, Cambridge, MA (Aug. 1991), in Lecture Notes in Computer Science, Vol. 523, pp. 73–101, J. Hughes (ed.).